SC-FDMA based Tropo-Scatter Modem Implementation in FPGA

A Project Report

submitted by

BIBIN BASHEER

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING,
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2017

THESIS CERTIFICATE

This is to certify that the thesis titled **SC-FDMA based Tropo-Scatter Modem Implementation in FPGA**, submitted by **Bibin Basheer**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Nitin Chandrachoodan
Project Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 22 June 2017

ACKNOWLEDGEMENTS

I thank the most benevolent and most merciful God almighty for all the blessings that has showered upon me throughout my life. I express my reserve gratitude to Dr. Nitin Chandrachoodan for his valuable guidance, constant support and encouragement for completing the project. I would like to thank HOD Prof. Devendra Jalihal for his support and IIT Madras for providing the facilities required for the completion of the project. I thank my organization DRDO and Director of NPOL, Kochi for sponsoring me to do M-Tech in IITM, Chennai. I would like to thank Sarnath, Janaki, Anand and Saravanan for providing support throughout my project related activities.I thank all my lab mates for their guidance and support.

Last, but most importantly I thank my mother, wife, daughter and my in laws for their encouragement and motivation without which it would not have been possible for me to finish my M-Tech Course.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

RTL	Register Transfer Level
HLS	High Level Synthesis
SC-FDMA	Single Carrier Frequency Division Multiple Access
OS	Operating System
DEAL	Defence Electronics Application Laboratory
DRDO	Defence Research and Development Organization
IP	Intellectual Property
RF	Radio Frequency
LwIP	Light Weight Internet Protocol
FPGA	Field Programmable Gate Array
EDA	Electronic Design Automation
HDL	Hardware Description Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
SDK	Software Development Kit
GNU	GNU's Not Unix
GCC	GNU Compiler Collection
JTAG	Joint Test Action Group
BSP	Board Support Package
IDE	Integrated Development Environment
FIFO	First In First Out
FSM	Finite State Machine
XPS	Xilinx Platform Studio
FFT	Fast Fourier Transform
CORDIC	COordinate Rotation DIgital Computer
RAM	Random Access Memory
API	Application Programming Interface
AXI	Advanced eXtensible Interface

HDF	Hardware Description File
XML	eXtensible Markup Language
POSIX	Portable Operating System Interface
QPSK	Quadrature Phase Shift Keying
FSK	Frequency Shift Keying
FDMA	Frequency Division Multiple Access
MCM	Multi Carrier Modulation
PAPR	Peak to Average Power Ratio
OFDM	Orthogonal Frequency Division Multiplexing
IFFT	Inverse Fast Fourier Transform
ICI	Inter Carrier Interference
ISI	Inter Symbol Interference
CP	Cyclic Prefix
IBI	Inter Block Interference
LDPC	Low Density Parity Check Code
FIR	Finite Impulse Response
BPSK	Binary Phase Shift Keying
DFT	Discrete Fourier Transform
ISR	Interrupt Service Routine
DMA	Direct Memory Access
SG	Scatter Gather
MAC	Media Access Control
TEMAC	Tri-mode Ethernet Media Access Control
SGMII	Serial Gigabit Media Independent Interface
PHY	Physical Layer
CFO	Carrier Frequency Offset
TCP/IP	Transmission Control Protocol / Internet Protocol
UDP	User Datagram Protocol
MTU	Maximum Transmission Unit
DAC	Digital to Analog Converter
ADC	Analog to Digital Converter
BRAM	Block RAM

ABSTRACT

KEYWORDS: SC-FDMA, Vivado HLS, Vivado IP Integrator, Xilinx SDK

The current trend in digital design is to accelerate the design and developmental cycles without compromising on verification. One of the key factor underlying the concept is to raise the abstraction layer from the traditional RTL level which are time consuming,error prone and difficult to debug. HLS tools are an attractive proposition for the rapid prototyping of the systems and bridges the gap between development times and time to market. HLS tools automatically transforms the algorithms written in C, C++, System C to RTL implementations.

In this thesis I am presenting a case study of using Vivado HLS tool and Xilinx Design Suites including Vivado IP Integrator and Xilinx SDK for the design and implementation of a wireless base band cum band pass SC-FDMA transmitter and receiver. In order to improve the performance of the hardware various optimization's like data flow pipelining, loop pipelining, array partitioning are used to convert the C code to RTL implementations. The challenge was to achieve the required performance through minimized iteration interval with less additional hardware overhead.

The transmitter and receiver IPs developed using Vivado HLS tools were tested standalone using the HLS tools and the IPs were exported to Vivado IP integrator where the full system including interfaces for RF transceiver's, Ethernet etc were built along with Microblaze softcore processor. The integrated system was tested using an application running on standalone OS in Microblaze softcore processor taking LwIP echo server as basis.

The hardware generated are successfully tested in VC707 evaluation board and finally integrated and tested successfully with customized board based on Vertex 7 series FPGA (XC7V585T-FFG1761-1) from DEAL, DRDO, Deharadun.

CHAPTER 1

INTRODUCTION

1.1 Introduction

The move to take new products, which are characterized by highly complex designs, to market as fast as possible made the EDA community to device strategies that reduces the design and development cycles without compromising on verification. The strategies adopted for design acceleration are *(i) design reuse* and *(ii) making higher abstraction level of the design*.The strategies to reduce design and development cycles in turn implies reduction in developmental costs.

Xilinx Vivado HLS is a tool for synthesizing digital hardware directly from a description of the system either in C/C++ or SystemC.This eliminates the need of designing the hardware in HDL languages like VHDL/Verilog. This high level design doesn't fix the hardware architecture as is done by HDL languages. The most important feature of the HLS tools are the designed functionality and its hardware implementations are kept separate and this provides great flexibility to the design community as various architectures can be explored and arrived at an optimum design.

Xilinx SDK is a part of Vivado IDE which provides a platform for creating fully functional software applications. The SDK includes GNU based compiler tool chain (GCC compiler, GDB debugger), JTAG debugger, flash programmer, drivers for Xilinx IPs and standalone BSPs and libraries for application specific functions. All of these features are accessible from within the Eclipse based IDE.

1.2 Motivation

Current research in the field of wireless communication is tending towards increasing the data rates with increased spectral usage efficiency. This is often accompanied by trying out various algorithms and optimization of these algorithms which the situation

demands. Sometimes the demand is to build customized communication systems which put forth specific requirements of the usage of specific algorithms.

The present case is to build a customized communication system for a defence application by DEAL, DRDO, Dehradun. The constraints in terms of the hardware and data rate requirements are defined by the user. The thesis presents the implementation of the specific algorithms which makes up the system on the hardware.

1.2.1 Pros and Cons of HLS

There are a wide range of HLS tools available which are differ by their ease of use and quality of the hardware derived.Some of the challenges that HLS tools must overcome (Donald.G.Bailey (2015)) are listed below.

- Software algorithms are sequential in nature whereas hardware operates concurrently. HLS must map the sequential algorithm onto concurrent hardware.
- Due to the sequential nature of software execution timing is implicit whereas hardware deals with timing constraints by controlling and synchronizing operations at the clock cycle level.
- Software supports fixed word lengths say 8,16,32 or 64 whereas hardware supports arbitrary precision data types like 5,11,15 etc depends on the computation being performed.
- Software supports dynamic memory allocation whereas hardware doesn't support, as local variables are stored in registers with distributed address spaces.
- Data transfer between various modules is through shared memory in case of software's whereas hardware depends upon the use of FIFO's , stream interfaces and associated handshaking signals for flow control.

Synthesis of an algorithm using HLS tools generally perform the following steps: data-flow analysis is being carried out as part of its algorithm synthesis to determine the type of operations that need to be performed; then resource allocation is done to determine how the resources on the hardware can be made use of in building the hardware followed by resource binding to determine the type of operations to be done by which hardware resource and finally scheduling which determines at what instant of time each operations will be performed to obtain the desired functionality.

Some of the key benefits in using HLS are listed below:

- Portability of the source code to any hardware which rarely involves restructuring of the code.
- HLS tools during its algorithm synthesis will analyze the structure of the code (loops, branches etc) to automatically extract and build the control path (FSM) whereas traditional RTL design requires explicit coding of the control path, which for complex designs it will be a herculean task.
- Data dependencies and the sequence of operation are analyzed by HLS tools to exploit parallelism which can be pipelined further to achieve the desired timing constraints.
- A pipelined architecture can be inferred from loops which involves less data dependencies between successive iterations.
- Iterations in a loop can be made parallel by loop unrolling which divides the loops to multiple hardware.

Design space exploration is accomplished through a combination of source code optimization and synthesis directives. Finding out the optimum design is generally a trade off between the speed and resources. Software profiling tools in HLS helps to identify processing bottlenecks, which enables to put more efforts on areas where it can potentially achieve the greatest gains. Design explorations at early stage can be achieved through HLS tools as it provides reasonably accurate estimate of resources without going for synthesizing the resulting RTL. In contrast,design exploration at RTL coding level will generally require considerable amount of time as it involves re-coding to change both the data and control paths and its error prone too. In HLS as the verification takes place at a higher level, simulations or verification's required for the generated designs are faster. But there is a necessity to validate the final design at the RTL level to ensure that the algorithm transformations are correct and HLS automatically generates RTL test benches from high level verification test benches.

Some of the key limitations in using HLS are listed below:

- Code restructuring is a must in realizing the hardware in order to improve performance.
- Treating HDL like a software leads to inefficient use of resources.
- Algorithms are based on pointers, whereas hardware implementations rely on arrays and array references which makes HLS finding it difficult to map to hardware and hence at many times a restructuring of the code is required.
- Recursive algorithms are very difficult to translate to hardware using HLS.

- Verification failure at the RTL level will be very difficult to analyze as the RTL code generated through HLS lacks human readability.
- Best algorithms for software realizations are not preferably the best suited for hardware implementations.

1.2.2 Design Flow Comparison between RTL and HLS

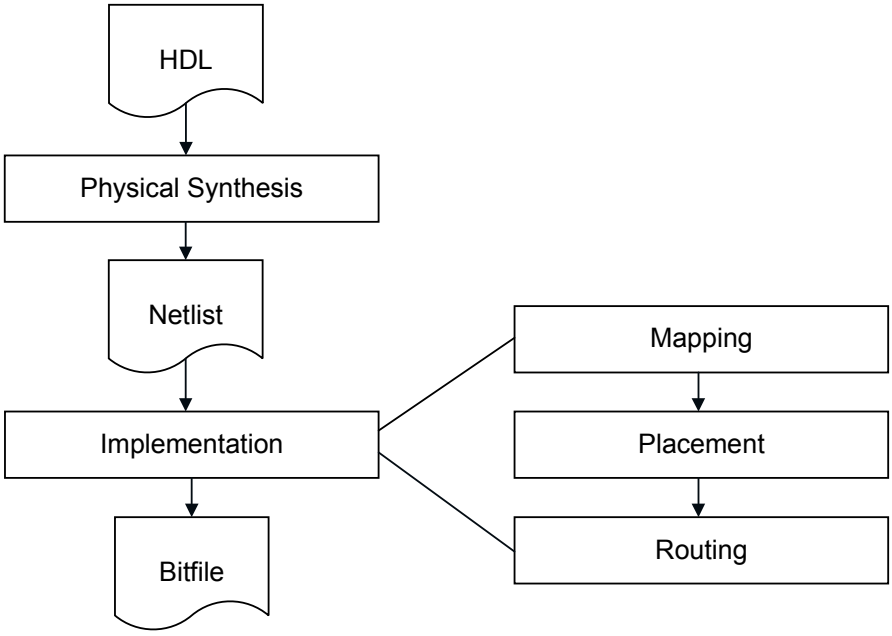

Figure 1.1: RTL Design Flow

The traditional RTL design flow is shown in Figure 1.1. The design entry comprises of files written in the HDL such as Verilog or VHDL. The design is described by RTL because any synchronous digital hardware circuit can be represented using Huffman model representation of an FSM wherein the data flows between hardware registers and the combinational circuits does the operation on these data.The functionality written in HDL's are verified by using test benches which are also written in HDL. The test bench provides inputs to the design and also receives outputs from the design. This process is called *Behavioural Simulation* where the functionality of the design is verified. The physical synthesis converts this description in hardware language into netlist which has gates, registers and wires. The physical synthesis which consist of technology mapping, placement and routing adds information's such as gate delays, wire length, location of gates to produce the final bit file. The bit file can be used to program the FPGA to perform required digital functionality.

HDL level description of a hardware consumes a major chunk of the time as it is written at a lower level of abstraction than algorithmic level. RTL's are written at a level of multiplexers,flip-flops *etc*.. In addition HDL is a concurrent programming language which makes them difficult to understand and code. HLS takes advantage of the situation through ease of programming thereby increasing productivity and ease of understanding.

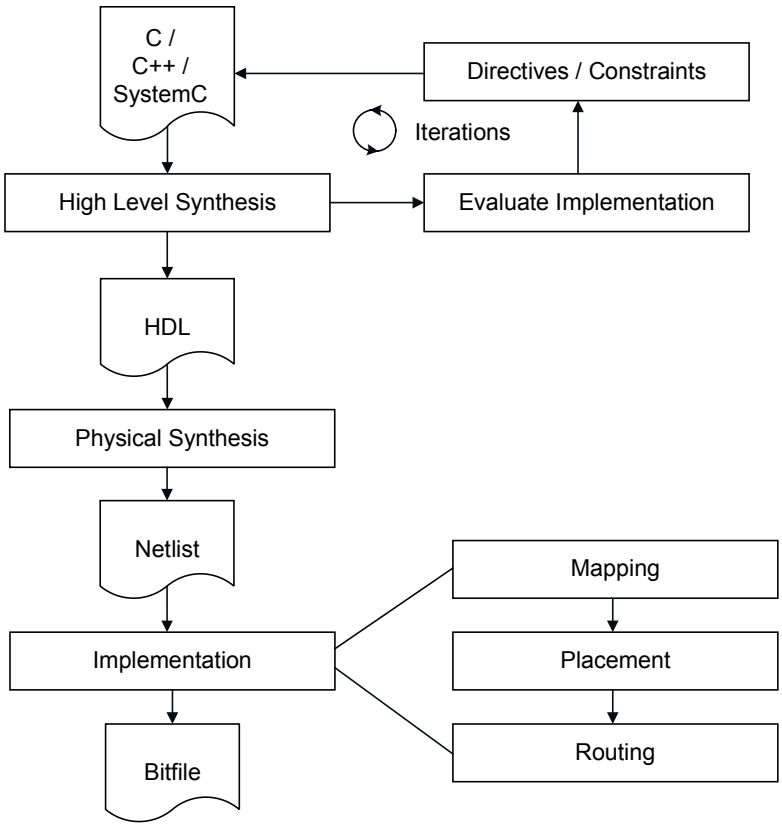

Figure 1.2: HLS Design Flow

HLS based design flow is shown in Figure 1.2. The design entry is in C/C++ or SystemC which allows the designer to describe an algorithm in a sequential manner. The algorithm written in high level languages are verified through a test bench which is also written in the same language and the process of functional verification is called C Simulation. After C Simulation produced the required result, the HLS synthesis tool analyzes and process the C based code with user specified constraints and directives and converts the C/C++ description into RTL which is called the C Synthesis or High Level Synthesis – the rest of the flow is similar to the traditional RTL flow.The design space exploration happens through the incremental changes provided by the user in the form of directives and hence different architectures are evaluated in due course.

Once the equivalent RTL model is produced, it can be further verified against the original C/C++/SystemC code via the process of *C/RTL Co-Simulation*.This process re-uses the original, C-based testbench to supply inputs to the RTL version generated by HLS, and check the outputs it produces against expected values.Importantly, this saves the effort of generating a new RTL test bench. The advantages of this flow are higher productivity, flexibility of design and architectural exploration which also helps to identify performance bottlenecks and area requirements at an early stage of design. The simulation at HLS based flow is much faster compared to the RTL level as the former allows to do so in C/C++ level. Error correction and design re-use at HLS based design is more easier and hence cost effective than RTL based design.

Once the design has been validated, and the implementation iterated to the point of achieving the intended design goals, it will be intended for integration into a larger system. This can be achieved directly through the use of packaging the outputs produced by Vivado HLS by means of *Export RTL*. Packaging to an IP helps to introduce HLS designs easily into other Xilinx tools, namely IP Integrator within Vivado IDE, XPS (for the ISE design flow), and System Generator.

Once the design has been integrated and implemented with HLS generated IPs and specific IPs provided by Xilinx like softcore processors and other peripherals, the whole hardware system can be controlled through application specific software's written on top of it through Xilinx SDK. The application in the form of elf (*executable and linkable format*) can be made to run on softcore processors and the system can be made to interface with external world.

Keeping this background in mind, the motivation for the present work is to use HLS, Vivado IP integrator and Xilinx SDK for the implementation of a specific wireless system.

1.3 Organization of the Thesis

The organization of the thesis will be as follows

Chapter 2 - describes the features of Vivado HLS, Vivado IP Integrator and Xilinx SDK that are being utilized for the development of modules for implementing the wireless communication system.

Chapter 3 - describes the basic concepts of a wireless communication specific emphasis on the SC-FDMA system.

Chapter 4 - mentions the algorithms that made up the whole system like FFT, Schmidl-Cox, CORDIC etc.

Chapter 5 - presents the results of implementation of the SC-FDMA system.

Chapter 6 - deals with the conclusions and future work.

CHAPTER 2

Background

The details present in this chapter contains information that are part of Xilinx (2016*a*) , Xilinx (2016*b*) and Xilinx (2017). But are reproduced here for easy reference.

2.1 Introduction to Vivado HLS

This section will cover topics including the specification of data types and its implications for synthesis, the creation of port and block-level interfaces, aspects of algorithm synthesis and the use of directives and constraints to influence the solutions produced by HLS.

2.1.1 Data Types

The specification of data type affects the resource utilization, timing performance, and power consumption. Under-specifying the word length compromises accuracy, while over-specifying leads to increased resources, inflated power consumption, and a sub-optimal maximum clock frequency. Vivado HLS supports both the native C/C++ data types plus the arbitrary precision data types for integer (for C/C++) and fixed point (for C++ only). It also supports the complex data type by including the header file "complex.h".

For the arbitrary precision fixed point data type W denotes the width of the word length and I denotes the bits specified for integer portion. So $W - I$ decides the bits for fractional representation. Q is a string which specifies the quantization mode and O gives the overflow mode. N specifies the number of bits in overflow wrap mode. The strings Q, O and N are optional. Vivado HLS also supports floating point data types and operations, as long as these maps to available Xilinx technology core libraries.

Language	Data Type	Description	Header
C	intN	N bit precision signed integer	include "ap_cint.h"
	uintN	N bit precision unsigned integer	
C++	ap_int<N>	N bit precision signed integer	include "ap_int.h"
	ap_uint<N>	N bit precision unsigned integer	

Table 2.1: Arbitrary precision data types for C/C++

Language	Data Type	Description	Header
C++	ap_fixed<W,I,Q,O,N>	signed fixed point number of I integer bits and W-I fractional bits	include "ap_fixed.h"
	ap_ufixed<W,I,Q,O,N>	unsigned fixed point number of I integer bits and W-I fractional bits	

Table 2.2: Arbitrary precision fixed point data types for C++

2.1.2 Interface Synthesis and Functions

The input arguments and return value of the designed top-level C/C++ function are synthesized into RTL data ports, each with an associated protocol. The RTL data ports can be either input, output or in-out bidirectional ports depending on whether data is read, write or both read and write into the ports respectively. Array arguments in top functions will translate to off chip RAMs . Arrays can be partitioned to increase the speed of access and by default are mapped to single port RAMs. The ports and protocol form a port interface. Port interfaces are used to communicate with other subsystems like the processor in the system. In addition to the port interfaces , block-level protocols and associated ports are used to coordinate the exchanges of data between subsystems.

2.1.3 Area/Resource

By default, Vivado HLS minimizes area, which implies time-sharing of hardware. This generally leads to increased latency and reduced throughput. Latency is defined as the number of clock cycles between applying an input, and achieving the corresponding output. Latency can be viewed at different levels of hierarchy and in the context of

loops, latency refers to the completion of all iterations of the loop and the term iteration latency is used when referring to a single iteration. The total latency is equal to the iteration latency, multiplied by the number of iterations of the loop (known as trip count). Latency can also be specified as a design constraint by the user, and the Vivado HLS tool optimizes the design wherever possible to meet the requirement.The Iteration Interval (II) is the number of clock cycles that separate the acceptance of inputs to the Vivado HLS design. Without applying directives, the initiation interval will be one cycle more than the latency, because the default behaviour of Vivado HLS is to optimize for area, resulting in a serial design. However the use of pipeline directive can reduce the iteration interval to much less than the latency of the design. This results in an increase in the area of the design, so there is a trade-off involved. Initiation interval corresponds directly to throughput. Throughput expresses the rate at which data can be passed through the system. The best possible initiation interval is 1, meaning that new input samples can be accepted on every clock cycle, in which case the throughput is equivalent to the clock rate. Higher levels of throughput can be achieved through use of partial loop unrolling, or by replicating a synthesized function.

2.1.4 Pipelining

In HLS, pipelining refers to the partitioning into sub stages of an arbitrary set of dependent operations. The objective for pipelining is to enable parallel processing, and thereby increase the throughput supported by the design. Pipelining can be applied as a directive in Vivado HLS, at the level of functions and loops.

2.1.5 Loops

Loops are basic constructs of any algorithm and expresses operations that are repetitive in nature. Several loop optimization's can be made using directives, enabling architectural variation exploration with almost no change required in the software code.The various optimization's performed in loops are

Loop Unrolling

It means that the hardware inferred from the loop body is created N times. Practically it can be less than N, depending on whether data dependency or memory operations are

present in the design. Advantage is throughput increase and disadvantage is the increase in hardware resource.

Partially Unrolled

This generally is a trade-off between rolled and unrolled architecture.

Merging of Loops

This directive can be applied to loops which are occurring one after the other in code and are having the same trip count. Advantages adds in creating the control path of the design as it reduces the number of states in the FSM derived.

Loop Flattening

It applies to nested loops where the overhead in additional clock cycles associated with entering and exiting the inner loops can be avoided which results in improved latency and throughput.

2.1.6 Arrays

As arrays represent storage they are synthesized into memories. The memories inferred are mapped to physical resources on the FPGA as Block RAM, or distributed RAM. Various directives are used to map these memories to physical memory resources. Some of the optimization's on arrays are discussed below.

Resource

It maps an array to a specific memory resource.

Array Map

It maps several small arrays to be combined into a single, larger array. Mapping can be either horizontal (arrays are concatenated to form an array with more elements), or vertical (array elements are combined, resulting in an array with longer words).

Array Partition

It maps the subdivision of a large array into a set of smaller ones. It increase the rate at which memory transactions can take place. In the extreme case, array partitioning will subdivide an array into individual register elements.

Array Reshape

This allows an array with many elements, each with short words, to be reshaped into an array with fewer, longer words.This directive reduces the number of required memory accesses.

Stream

If the array element access is sequential not random, then this directive can be made use of. It reduces the number of ports generated.

2.1.7 Exporting from Vivado HLS

Designs can be exported from Vivado HLS to permit easy integration of Vivado HLS IP with other development tools. IPs are exported from HLS to the IP-XACT format, which allows the module to be integrated into a Vivado IP Integrator design. This results in a zip folder residing in the ip sub-folder under impl folder of the respective solution, and represents the IP catalogue package. It also contains API's required to use the IP in Xilinx SDK environment. This format allows easy sharing and distribution of IP across all platforms.

2.2 Introduction to Vivado IP Integrator

The Vivado IP integrator tool allows to create complex subsystem designs by instantiating and interconnecting IP cores from the Vivado IP Catalogue which contains Xilinx IP's , third party IP's and user IP's. Using the repository manager provided by the tool, we can add user developed IP's especially in Vivado HLS to the current project or can be added permanently to the user IP portion of the Vivado IP catalogue. Various design flows of the FPGA are provided as separate functions on the Vivado IP integrator tool. Even the RTL level designs can also be packaged to an IP using the tool and it will provide an easy and straight forward approach to the incremental building of a complex design. The tool provides different options to optimize the foot prints of the IP's implementation on FPGA which improves operating frequency, performance, or area and can be specified such that a suitable balance between these three metrics is achieved. This is done very easily in Vivado using a configuration wizard.

In a processor integrated system, the IP's are interfaced with each other generally on an AXI bus which are connected through an AXI interconnect. AXI interconnect can be considered as a switch in computer networks. Currently the standard is AXI4 and there are three variants of bus protocols namely

AXI4

It is provided for memory-mapped IP's, and having the highest performance with an address is supplied followed by a data burst transfer of up to 256 data words. **AXI4-Lite** It is a simplified link supporting only single data transfer per connection (no bursts). It is also memory-mapped. In this case an address and a single data word are transferred.
AXI4-Stream
This data streaming is provided for non memory mapped IP's and supports high-speed streaming data with burst transfers of unrestricted size.

2.2.1 AXI Architecture

The AXI protocol supports burst based transactions, with each containing address and control information on the address channel. Several AXI masters can be connected to several AXI slaves through an AXI interconnect. An AXI master transfers data to an AXI slave through the AXI interconnect using a write data channel (or a read data channel from slave to master). The write transactions in particular have an additional write response channel, as all data flows from master to slave and as such this is used for the slave to signal completion of a write transaction. The following figures demonstrate communication between AXI master and slave. The Figure 2.1 shows the write channel

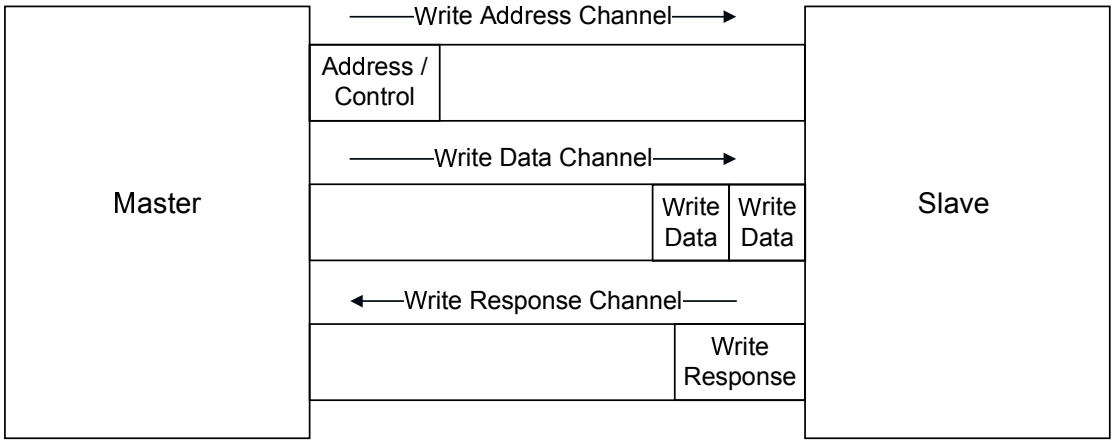

Figure 2.1: AXI4 write channel architecture

architecture where address and control data is passed from master to slave before a burst of data is transmitted, and a write response signalled following completion.The Figure 2.2 shows a read transaction, with address and control data transmitted to the slave before a burst of read data is transmitted to the master.

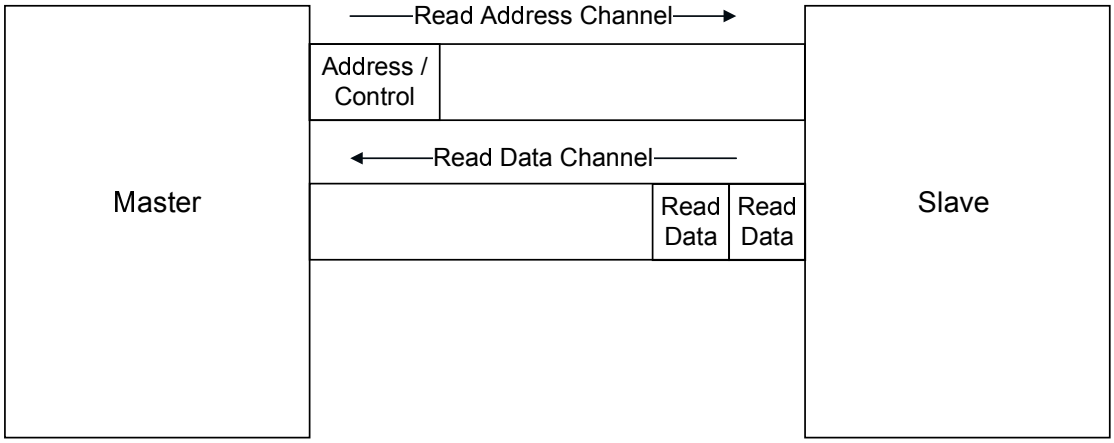

Figure 2.2: AXI4 read channel architecture

2.2.2 AXI Transactions

Write-Burst Transaction

Suppose we need to write burst of data to an address A. The master drives the slave and transaction begins with the sending of address and control information via the signal AWADDR. Following confirmation of a valid address with AWVALID, a signal is sent to confirm that the system is ready for the transaction on AWREADY. The master then sends the data blocks in the burst of data to the slave on the WDATA signal, with the final data item being indicated through the WLAST signal going high and confirming the completion of the transaction. The master also sends various control signals regarding data bursts.

Read-Burst Transaction

In case of a read-burst transaction using AXI4 for data being read from an address A, the slave is driven by the master through transmission of address and control information in signal ARADDR. ARVALID goes high signalling a valid address and the system is confirmed ready for transmission with the signal ARREADY. Data blocks are read from address A via the signal RDATA, and as before the final data block is indicated via signal RLAST. The RVALID signal kept low by the slave until read data is available.

2.3 Introduction to Xilinx SDK

Xilinx SDK provides an environment for creating software platforms and applications for Xilinx processor cores.It works with hardware designs generated with Vivado.

SDK is provided with

1. Reference Software Applications
   Applications like lwip echo server which we made use of for building this project.
2. XMD
   Xilinx Microprocessor Debugger is debug agent used to communicate with Xilinx embedded processors.
3. XSDB
   Xilinx System Debugger is a command-line interface to debug the Xilinx hw_server.
4. FPGA programmer
   Used to program the Xilinx FPGA with the bitstream.
5. Flash programmer
   Used for burning bitstream and software application images into external parallel NOR Flash devices.
6. Bootloader generator
   Used for automatically bootloading your embedded software applications from parallel Flash.

The two main terminologies used in Xilinx SDK are hardware platform and software platform.The hardware platform is the embedded hardware design that is created in Vivado and exported in the form of an HDF/XML file through the use of export hardware wizard. Once the hardware platform is identified and imported, we create the software platform. A software platform is a collection of libraries and drivers that form the lowest layer of application software stack. The software applications must run on top of a given software platform, using the provided API's. Therefore before creating and use software applications in SDK, a software platform project must be created. SDK includes the following two software platform types. They are

1. Standalone OS
   It is a simple and single-threaded environment that provides basic features like standard input/output and access to processor hardware features. In this project we extensively used this software platform.

2. Xilkernel
   A simple and lightweight kernel that provides POSIX-style services such as scheduling, threads, synchronization, message passing, and timers.

2.4 Basics of Wireless Communication System

Wireless communication involves the transmission of information over a distance without the help of a wired medium. It incorporates all methods of connecting and communicating between two or more devices using a wireless signal through wireless communication technologies and devices. A basic block diagram of a communication system is shown below.

The digitized source produces the binary data to be transmitted. The data which is

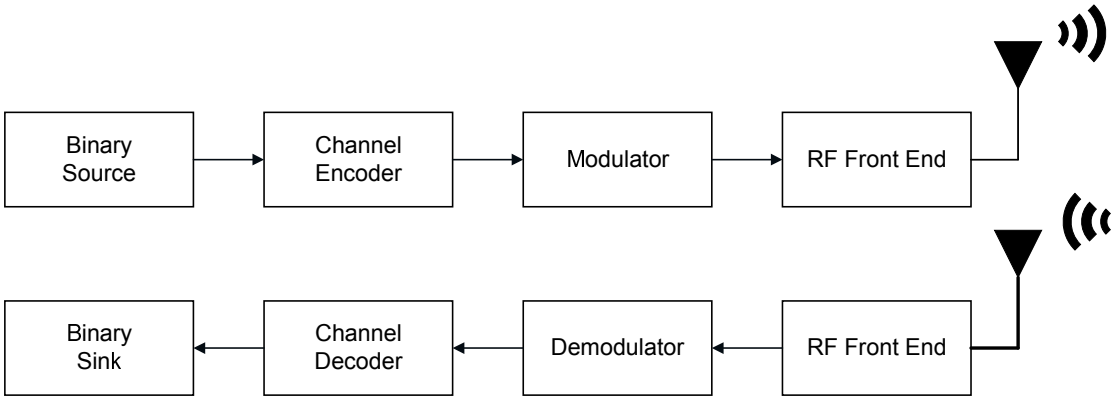

Figure 2.3: Basic block diagram of a wireless transceiver

transmitted through the channel suffers errors due to interference and noises and to mitigate the effects of these, channel encoding is used which enables forward error correction at the receiver. Generally system uses convolution encoding techniques or linear block codes for channel encoding. The encoded data is modulated using digital communication techniques like QPSK, FSK etc (base band signals) and is translated to radio frequency (band pass signals) by RF front end and is transmitted to antenna.The received RF signal suffers from addition of noise and fading due to wireless channel. The received signal traverses multiple paths, which causes fading. The RF front end at receiver translates band pass signals to base band and converts signal from analog to digital form. The data is decoded and demodulated to retrieve the information back.

The path followed by transmitted signal to reach receiver are of two types.*(i) Direct Path* - The transmitted signal reaches the receiver directly and the components of signal that are present are called direct path components.*(ii) Multi Path* - The transmitted signal traverses through different directions undergoing different phenomenon like reflection, diffraction, scattering and arrive at the receiver shifted in amplitude, frequency and phase with respect to the direct path component and such a path is called multi path

and the components present are called multi path components.

Multiple access schemes are used to allow multiple users to share simultaneously a finite amount of radio spectrum. There are different multiple access schemes and one method is FDMA. In FDMA, the spectrum is divided into narrow bands and each user is allocated with a specific channel. Hence a wide band signal which suffers distortion as the signal bandwidth is greater than the coherent bandwidth due to frequency selective fading of the wireless channel is converted to narrow bands and hence the channel response to these narrow band signals will be flat fading or provides no distortion as now the coherent bandwidth is more than signal bandwidth. This forms the basic principle behind the FDMA.

In order to mitigate the effect of fading, MCM schemes are employed. In MCM , the high data rate signal is divided into a N low data rate signals and each one is modulated on a separate sub-carriers. The final signal for transmission will be the algebraic sum of theses modulated sub-carriers. This technique is employed in OFDM communication where the N sub-carriers are orthogonal to each other. The modulation of low data rate signals on N sub-carriers is equal to taking the signal samples or symbols and performing an IFFT operation on these samples. But the disadvantage of such a scheme is the high PAPR which is directly proportional to the number of sub-carriers N.The high PAPR in an OFDM system arises due to IFFT operations as data symbols across sub-carriers can add up to produce a high peak value. As the power amplifiers practically are non linear having very restricted linear region, these peak variations will result in distortions and the orthogonality of the sub-carriers can be lost. Hence, high PAPR in OFDM systems results in amplifier saturation leading to non linear distortion and ICI. Inorder to tackle the issue of high PAPR another technique has been introduced known as SC-FDMA. This technique is used in the up-link of 4G communication systems.So an OFDM system is transformed to a SC-FDMA system by putting an N point FFT block in front of an N point IFFT block for sub-carrier modulation. Thus the effect is cancelled with each other and gets a single carrier modulated signal. But single carrier modulation scheme suffers from ISI as the delay spread of the channel is more compared to the symbol duration. Hence the transmitter of SC-FDMA employs an M point FFT followed by an N point IFFT where $M < N$, which mitigates the effects of high PAPR and ISI.

CHAPTER 3

SC-FDMA

This chapter gives information about the various algorithms that were used to implement the system using Vivado HLS. Detailed description of the algorithms and synthesis of algorithms to RTL using Vivado HLS tool was described in Salaskar (2016).

3.1 System Description

The base band transceiver block diagram of the system is show in the Figure 3.1. The

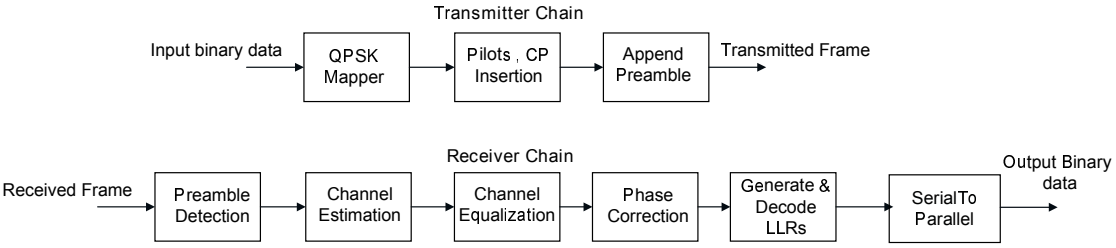

Figure 3.1: Block diagram of a SC-FDMA transceiver system

input binary data to be transmitted is passed through a QPSK mapper. Then the CP and the pilots are inserted and finally appended with the Preamble to form the final transmitter frame. The CP is added to avoid IBI and pilots are inserted in the data to estimate the channel at the receiver for equalization. The frame is transmitted through channel by converting to analog signal. The frequency translation to radio frequency and digital to analog conversion is performed by RF transceiver modules (AD9364). The signals are further conditioned in the RF front end where they will be further amplified and transmitted.

The received RF signal suffers from addition of noise and fading due to wireless channel. The received signal traverses multiple paths, which causes fading. The RF transceiver modules at the receiver translates from radio frequency to base band frequency and converts signal from analog to digital form. The synchronization between the transmitter and receiver happens in the preamble detection, where the receiver detects frame in the coming data and tries to equalize the effect of channel using the known pilots inserted in

the transmitted data. The QPSK de-mapper converts the equalized data back to original message data

3.2 System Specification

The system specification requirement was to achieve a throughput of 20 Mbps with LDPC code rate of $2/3$ and bandwidth of 20 MHz. But LDPC modules are not implemented in the present system.

3.2.1 Frame Structure

The frame structure used in the SC-FDMA system is shown in the Figure 3.2. Each

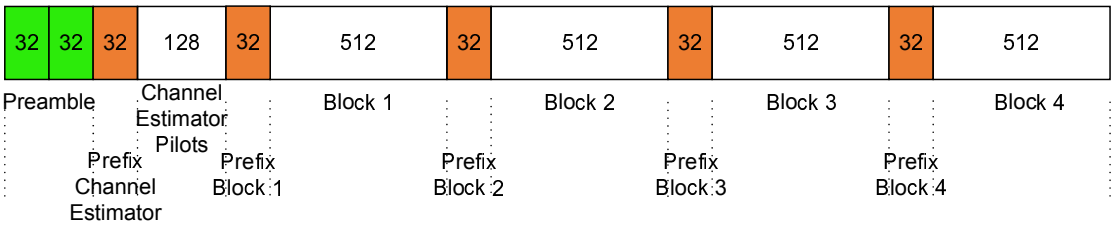

Figure 3.2: Frame Structure of SC-FDMA

SC-FDMA frame consists of

- Preamble of 64 samples for packet detection using Schmidl-Cox synchronization method.
- Pilot symbols (128 pilots + 32 CP) for channel estimation and frequency offset correction.
- 4 message blocks of which each composed of
  - 480 complex QPSK symbols
  - 32 pilot symbols
  - 32 CP symbols

Hence the total frame size is 64 + (128+32) + 4 * (480+32+32) = 2400 samples. The information bits available in each frame assuming an LDPC code rate of 2/3 is

- 4 blocks per frame
- 480 QPSK symbols per block
- 2 bits per QPSK symbol
- $2/3$ LDPC code rate

Hence total number of message bits = 4 * 480 * 2 * 2/3 = 2560 bits. Therefore assuming a sampling rate of 20 MHz for the transmit DAC's I/Q output, this will result in 21.33 Mbps. So if we are using a clock frequency of 20 MHz we can process the entire frame in 2400 cycles. Tn this project we are choosing a 100 MHz clock and hence we can spend at most 12000 cycles to process an entire frame.
CP refers to prefixing of a symbol with the data at the end. It is used instead of a guard interval between two symbols. The addition of cyclic prefix mitigates the effects of channel fading and ISI thereby increases the bandwidth. This repetition of end symbols allows us to visualize the linear convolution (channel modelled as an FIR filter) as a circular convolution. This also reduces IBI between subsequent message blocks.
Modulation is the process of facilitating the transfer of information over a medium. QPSK is the digital modulation scheme used in this communication system. In QPSK, the phase of the carrier is modulated in accordance with the information message bits. Two message bits are combined which results in four combinations say 00,01,10,11 and each combination represents a certain known phase which is used to modulate the carrier wave. The de-modulator in the receiver must determine the phase of each symbol and map it back to the corresponding message bit. QPSK encodes two bits per symbol and has double the data rate compared to BPSK. The constellation diagram of the QPSK is shown in Figure 3.3.

3.2.2 Schmidl Cox Algorithm

Proper synchronization between transmitter and receiver ensures correct reception of data which in turn means that the receiver is able to lock exactly to the beginning of the transmitted signal. In other words, the time must be synchronized between the transmitter and the receiver. This forms the block of preamble detection which uses the

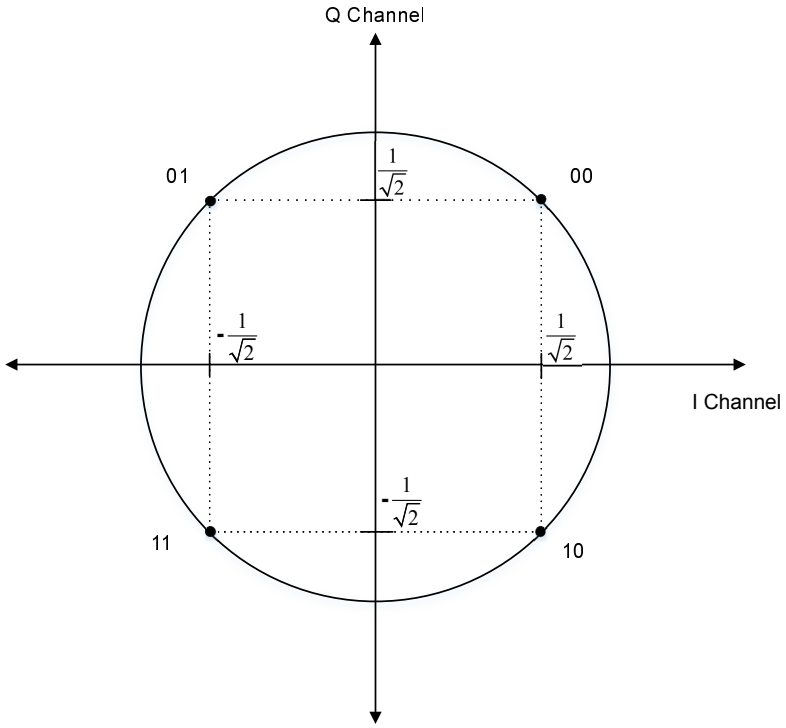

Figure 3.3: QPSK signal constellation mapping

Schmidl-Cox algorithm. The Schmidl-Cox algorithm is a time synchronization algorithm proposed by Schmidl and Cox (1997) for use in OFDM based systems. In this method,a specially generated preamble is prefixed to the data to be transmitted, which is used by the receiver for synchronization. The variations in the channel often make synchronization necessary. The Schmidl-Cox preamble has pseudo-random complex Gaussian noise on the odd frequencies, while having zero amplitude on even frequencies. This composition of the transmitted preamble ensures certain properties which are used for synchronization. The preamble is further processed in the receiver to achieve time synchronization.

The preamble is added in the final stage of transmitter and is detected at the first stage of the receiver. Both in-phase and quadrature components are used in the detection process. The preamble consists of two highly correlated sections of equal length. The in-phase component is symmetric about its center and the quadrature component is anti-symmetric about its center.

Detection Method The Schmidl-Cox based threshold detection method finds the points at which the S signal crosses a predefined threshold value, first above then below. The centre of the peak which is synchronization point is obtained by averaging these two

points. The S signal is defined as

$$S(d) = \frac{(P(d))^2}{(R(d))^2}$$

where, cross-correlation value P is defined as

$$P(d) = \sum_{m=0}^{L-1} r^*_{d+m} r_{d+m+L}$$

auto-correlation value R is defined as

$$R(d) = \sum_{m=0}^{L-1} |r_{d+m+L}|^2$$

3.2.3 Channel Estimation

Due to the dynamic nature of the channel owing to various factors , it is of at-most importance to carry out channel estimation to determine the channel transfer function $H(f)$. The channel estimation pilots present in the frame structure are used for this purpose. These pilots form a Zadoff-Chu (ZC) sequence (Zepernick and Finger (2013)) and are extracted after preamble detection.

Zadoff-Chu (ZC) sequence ZC sequences are complex-valued and have constant amplitude whereby cyclically shifted versions of the sequence imposed on a signal result in zero correlation with one another at the receiver. These sequences exhibit the useful property that cyclically shifted versions of itself are orthogonal to one another, provided that each cyclic shift, when viewed within the time domain of the signal, is greater than the combined propagation delay and multi-path delay-spread of that signal between the transmitter and receiver.

The envelope of the channel impulse response is obtained by correlating the pilots in the receiver. A DFT based method is used to estimate the channel from the envelope. As the symbol duration is longer than the duration of channel impulse response (Kang *et al.* (2007)),the estimated channel has most of the power concentrated in first few coefficients. The DFT based channel estimation reduces the noise power that exists outside the first few coefficients. Hence most of the information about the channel impulse response is limited to length of 20. So first 20 samples have been taken for channel

estimation purpose.

The algorithm used for channel estimation and equalization uses FFT.

3.2.4 FFT

The Fourier Transform converts a time domain signal $x(t)$ to a function $X(f)$ in frequency domain:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-2\pi ft}dt$$

The discrete counter part of the Fourier transform is DFT.

$$X(k) = \sum_{0}^{N-1} x(n)e^{\frac{-2\pi nk}{N}}$$

The FFT algorithm reduces the complexity of computing DFT from $O(N^2)$ to $O(N\log(N))$. FFT algorithm can be implemented in *(i) Decimation in Frequency* and *(ii) Decimation in Time*. Decimation in Frequency has been utilized in this work. The algorithm starts with splitting of input data sequence into two summations of which one involves the sum over first N/2 data points and second sum over last N/2 data points. Then the algorithm is recursively applied to each term until each DFT's length is 1. This recursive deconstruction of the DFT makes the computational time of $O(N\log(N))$ (Proakis and Manolakis (2006)). It requires $N/2\log_2(N)$ complex multiplications and $N\log_2 N$ complex additions. This algorithm computation can be performed in-place in order to reduce the storage requirement.

3.2.5 Channel Equalization

Multi-path propagation in wireless communication results in fading and delay dispersion's which leads to ISI in the transmitted signal.The effects of fading is reduced by the implementation of channel coding in addition to the cyclic prefixes. The ISI equalizers are introduced in the receivers in order to improve the quality of the signal. The equalization techniques are adopted to reverse the distortions produced by the channel. The delay dispersion corresponds to the frequency selectivity in frequency domain. Due to presence of ISI effect the transfer function is not constant over bandwidth of interest. The equalizers will provide an inverse effect of the channel to the transmitted

signal thereby nullifies the effect of channel. Mathematically it can be expressed in the following way:

$$H(f)E(f) = constant$$

where $H(f)$ is the transfer function of the channel, $E(f)$ is the transfer function of the equalizer. Ideally we should have,

$$E(f) = \frac{1}{H(F)}$$

where $E(f)$ should be exactly the inverse of the channel transfer function $H(f)$ to obtain a flat response. However, due to the inversion of $H(f)$, infinity gains are produced when $H(f)$ values are close to 0, preventing the desired constant output. Zero-Forcing is the most basic equalizer and its coefficients try to enforce a constant transfer function in frequency domain once applied to the channel. Zero-Forcing equalizer is ideal for elimination of ISI. But, this kind of equalization produce noise enhancement in the spectral nulls (deep fading) due to the form of the transfer function of the equalizer, which is $E(f) = \frac{1}{H(f)}$. From above equation, it can be deduced that the equalizer amplifies strongly (infinity gain) those frequencies with a small value of the channel transfer function, enhancing the noise too.

3.2.6 CORDIC

CORDIC stands for COordinate Rotation DIgital Computer and belongs to the class of shift and add algorithms. The CORDIC algorithm was presented by Volder (1959) as a method of calculation of trigonometric and hyperbolic functions. Subsequently, Walther (1971) generalized the algorithm to calculate elementary functions such as multiplication, division, trigonometric ratios, square roots, etc. The fundamental advantage of the CORDIC algorithm is that most basic implementations only requires addition/subtraction, table look up and bit shift. This makes it particularly suitable for implementation on FPGA's, where the multipliers are a scarce resource. CORDIC can be either used in *(i) Rotation Mode* , the rotation of an input vector by a given angle α and *(ii) Vectoring Mode* to calculate inverse tangents of rotation of an input vector V.For details on the description of the algorithm refer Salaskar (2016).

CHAPTER 4

Implementation of the System

The detailed description of the implementation of the components of transmitter and receiver systems using Vivado HLS has been mentioned in Salaskar (2016). For the implementation of the system on specific hardware (customized Virtex 7 FPGA board from DEAL), re-synthesis of the existing HLS code has been done to ensure that the throughput requirements and device specific constraints are met. Along with this we briefly touch upon the results obtained after applying various types of optimization's that were applied to each of the components of the transmitter and receiver.

The synthesized transceiver modules are integrated with a Microblaze softcore processor based system which provide ease of programmability, debugging and if necessary portability to other FPGA hardware platforms. Xilinx provided peripheral IPs like ethernet,UART etc are also integrated with system which provides in-situ configuration management and ease of maintenance. In future, the implementation of a parametrized communication system along with microblaze softcore processor and ethernet helps to reconfigure the system depends on user requirements. It is also more easy to integrate the RF transceiver AD9364 modules, as Analog Devices provides an API in C for configuration of its registers.

The target for our implementation was to achieve an overall transceiver throughput of 20Mbps. Given that each frame contains 2560 valid information bits, every new frame has to be processed within 128μs, even though the initial latency of a frame could be larger than this amount.

Based on synthesis results of the basic blocks, a target clock frequency of 100MHz was chosen for the design. This implies that a frame of 2400 symbols would take 12,000 clock cycles (10ns per clock), and we use this figure as our target for the *initiation interval* in the synthesis steps that follow.

Algorithm 1 Transmitter

1: **procedure** SCFDMA_TX($data, Frame\_Tx$)

Input: $data$ 160 words of 16 bit each

Output: $Frame\_Tx$ of length 2400 Complex numbers

2: $wimaxencoder(data, Mapped\_Data)$

3: Append $preamble$ to $Frame\_Tx$

4: AssembleLoop:

5: **for** $i = 1 \rightarrow NO\_BLOCKS$ **do**

6: Append $CyclicPrefix(i)$ to $Frame\_Tx$

7: Insert $Pilots$ and $Mapped\_Data$ in $Symbol(i)$

8: Append $symbol(i)$ to $Frame_Tx$

9: **end for**

10: **end procedure**

4.1 Transmitter

The algorithm 1 describes the $scfdma\_tx$ function. The transmitter takes $data$ of length 160 of 16 bits each (2560 bits total) and produces $Frame\_Tx$ of length 2400 complex symbols with 16 bits I and Q. The $wimaxencoder$ function performs QPSK mapping. It takes 80 words of 16 bits and produces QPSK mapped data of length 960. The function $wimaxencoder$ is invoked twice in order to encode 2560 bits of data into 1920 QPSK complex values. The $Frame\_Tx$ is assembled by prepending the $preamble$. The $AssembleLoop$ affixes the cyclic prefix and inserts the pilots in appropriate location.

The algorithm 2 performs QPSK mapping. The $ReshapeLoop$ reorganizes the input data $data$ into $CodeWord$. The $CodeWord$ has 16 locations of 80 bit each. 16 locations are filled by rearranging $data$. The $QPSKMappingLoop$ takes two bit of data and maps it into complex symbol as follows

$(0, 0) \rightarrow (+, +)$,

$(0, 1) \rightarrow (+, -)$,

$(1, 0) \rightarrow (-, +)$,

$(1, 1) \rightarrow (-, -)$.

Algorithm 2 QPSK Mapper

1: **procedure** WIMAXENCODER($data, Mapped\_Data$)

Input: $data$ 80 words of 16 bit each

Output: $Mapped\_Data$ of length 960 Complex numbers

2: ReshapeLoop:
3: **for** $i = 1 \rightarrow N$ **do**
4: Reshape the $data$ to Z bits in a word $CodeWord[i]$
5: **end for** ▷ QPSK mapping
6: $QPSK(0) \leftarrow (IS2, IS2)$
7: $QPSK(1) \leftarrow (-IS2, IS2)$
8: $QPSK(2) \leftarrow (IS2, -IS2)$
9: $QPSK(3) \leftarrow (-IS2, -IS2)$
10: $index \leftarrow 0$
11: $ap\_uint < 2 > temp$;
12: QPSKMappingLoop:
13: **for** $k = 1 \rightarrow N$ **do**
14: **for** $l = 1 \rightarrow BLOCK\_SIZE$ **do**
15: $temp[0] \leftarrow CodeWord[k][l]$
16: $temp[1] \leftarrow CodeWord[k][l + 1]$
17: $Mapped\_Data(index) \leftarrow QPSK[temp]$
18: $l \leftarrow l + 2$
19: $k \leftarrow k + 1$
20: $index \leftarrow index + 1$
21: **end for**
22: **end for**
23: **end procedure**

The $ReshapeLoop$ gets synthesized as a memory which has write port which is 16 bits while the read port is 80 bits in length. The top-level function of the transmitter is `Transmitter` which has function prototype of

```
void scfdma_tx(
    ap_uint<16> data[160],
    ComplexData Frame_TX[2400]);
```

Optimizations

Table 4.1 shows the result of converting from C to RTL using the Vivado HLS synthesis system. Note: *LUT*s are the look-up tables that are the combinational building blocks

inside FPGAs, *FF*s are flip-flops, *BRAM_18K* are 18 Kbit block RAMs, *DSP48E* are the multiplier/DSP macros in Xilinx FPGAs. T_L refers to the latency of the loop, while II refers to the iteration interval, which is the earliest time at which the next iteration can be started (both are in number of clock cycles).

The unoptimized design uses 16 BRAMs 7 of them are used for QPSK mapper, 2 of them are used for AXI interface while another 7 are being used by the rest of the transmitter logic. Out of 7 BRAMs used for transmitter logic the QPSK mapper output used 2 BRAMs each for real and imaginary parts, the pilots use 1 BRAM each for real and imaginary parts and information to store the location of pilots to be inserted uses 1 BRAM. Hence in total 16 BRAMs are used. The QPSK mapper takes 3248 clock cycles as both T_L and II . The pilot insertion logic and preamble appending logic takes about 13750 cycles for latency and 13751 cycles for II.

We have to apply *dataflow* and *pipeline* optimization to improve the latency and throughput of the design. The overall structure of the transmitter is well suited to the idea of a dataflow operation where each block commences immediately on receiving sufficient data and transfers its output to the next block in the chain. This permits multiple hardware blocks to operate in parallel.

The Figure 4.1 shows the result of dataflow pipelining of $Transmitter$ function.

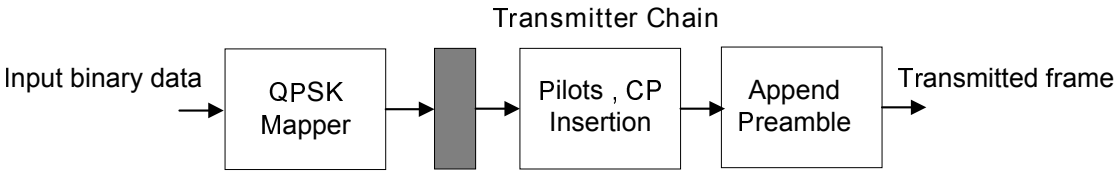

Figure 4.1: Transmitter after dataflow pipelining

The transmitter is dataflow pipelined using the pragma

```
#pragma HLS DATAFLOW
```

We see that number of BRAMs increase by 4. The additional 2 BRAMs each for real and imaginary parts of QPSK mapped data are used, which helps pilot insertion logic to operate in parallel, which also cuts down the II to about 7250 cycles.

The $AssembleLoop$ are pipelined using the pragma

```
#pragma HLS PIPELINE
```

	BRAMs	DSP48Es	FFs	LUTs	T_L	II	T_{clk} (ns)
Before Optimizations	16	0	1146	3621	13750	13751	7.38
After Data-flow pipelining	20	0	1195	3618	13751	7250	4.23
After all Optimizations	20	0	1199	3618	8921	6502	7.38

Table 4.1: Transmitter results before and after data flow & pipeline optimizations

From the results after pipelining the loop we see that II has come down to 6502 cycles.

The result shows that the transmitter is able to achieve our target constraint. Since our goal is to have an overall transmitter/receiver pair that can operate at the desired throughput, no more efforts has been put in to further increase the transmitter throughput. In the next section we look at the preamble detection or packet detection followed by receiver design which is considerably more complex.

4.1.1 Preamble detection

Algorithm 3 scfdma_pktdet

1: **procedure** SCFDMA_PKTDET($input\_data, out\_data$)

Input: $input\_data$ is input stream of data

Output: out_data are the data frame synchronized with packet detection

2: Initialize the $SregReal$ and $SregImag$
3: PackDetectLoop:
4: **for** $i = PKTBUFSIZE - 1 \rightarrow 1$ **do**
5: Update $SregReal$,$SregImag$ and insert new $input_data$
6: $old\_0.real() \leftarrow SregReal[1]$
7: $old\_0.imag() \leftarrow SregImag[1]$
8: $old\_32.real() \leftarrow SregReal[33]$
9: $old\_32.imag() \leftarrow SregImag[33]$
10: $old\_64.real() \leftarrow SregReal[65]$
11: $old\_64.imag() \leftarrow SregImag[65]$
12: $(n)ewpkt$=TIMESYNC$busy, old_0, old_32, old_64$
13:

The algorithm 3 describes the $scfdma\_pktdet$ function. The goal of this function is to detect the start of a new packet (using the Schmidl-Cox algorithm), and once

detected, to stream out data to the further blocks down the line in the receiver.

This function initializes the $SregReal$ and $busy$ to false. The $PackDetectLoop$ shifts in the new $input\_data$ and calls the $TimeSync$ function. The $TimeSync$ function takes the content of shift register and computes auto and cross correlation values to find the packet location (as per the Schmidl-Cox algorithm described in the previous chapter). The out_data streams the data based on start of packet in $input\_data$ and $newpkt$ variable tell if the packet is detected or not.

The $TimeSync$ function computes the auto and cross correlation of which the description was given in the previous chapter on the Schmidl-Cox method.

These auto and cross correlation values are used in checking the condition for the relation to cross the threshold value in comp block. It is also checked that the ratio remains above the threshold for some values which ensures that the threshold crossing is not due to some noise but the actual preamble is detected. The remaining frame is received and is passed on to the $memstore$ for extraction of preamble,pilots and data symbols from the data stream.

Optimization

The $scfdma\_pktdet$ function has two arrays "Sreg.Real" and "Sreg.Imag". These arrays are used to shift in the received data in for preamble detection. The $TimeSync$ function uses 0^{th}, 32^{nd} and 64^{th} values of this shift register. In order to access these values the array is partitioned completely. This is accomplished using the pragma

```
#pragma HLS ARRAY_PARTITION variable=Sreg.Real complete dim=1
#pragma HLS ARRAY_PARTITION variable=Sreg.Imag complete dim=1
```

The $PackDetectLoop$ is also pipelined. The $TimeSync$ function is pipelined in order to process one sample in one clock cycle.

	BRAMs	DSP48Es	FFs	LUTs	T_L	II	T_{clk} (ns)
Before Optimizations	4	20	554	456	158	159	9.78
After Optimizations	0	20	2600	414	4	1	8.51

Table 4.2: Preamble detection module before and after optimizations

4.2 Receiver

Algorithm 4 Receiver

procedure SCFDMA_RX($input\_data, data\_out$)
Input: $input\_data$ is input stream of data
Output: $data\_out$ are demodulated bits after receiving one frame it has a length of 2560
    MEMSTORE($input\_data, extracted\_pilots, extracted\_data$)
    CHANNEL_ESTIMATION($extracted\_pilots, est\_channel$)
    CHANNEL_EQUALIZATION($extracted\_data, est\_channel, equalized\_data$)
    GENERATELLRS($equalized\_data, LLR$)
    LLRLoop:
    for $i = 1 \rightarrow NO\_BLOCKS$ **do**
        for $j = 1 \rightarrow BLOCK\_LENGTH$ **do**
            if DataLoc[m]==i **then**
                $LLR[k] \leftarrow PhCorrectData[j][i].Real$
                $k \leftarrow k + 1$
                $LLR[k] \leftarrow PhCorrectData[j][i].Imag$
                $k \leftarrow k + 1$
                $m \leftarrow m + 1$
            end if
        end for
    end for
    HARDDECODELLRS($LLR, Decoded\_bits$)
    SERIALTOPARALLEL($Decoded\_bits, data\_out$)
end procedure=0

The algorithm 4 describes the receiver flow. The first block in the receiver chain is packet detection and is kept outside the receiver main module. The packet detection module will be always active and once it detects the packet it will raise a high signal and then data will be stored in $memstore$ and is copied to subsequent modules in the receiver chain. This way receiver modules will be held in active state only when packet detection happens. The $scfdma\_pktdet$ func-

Module	BRAMs	DSP48Es	FFs	LUTs	T_L	II	T_{clk} (ns)
memstore	0	0	38	86	2318	2318	6.42
Channel_Estimation	28	22	1931	2181	39102	39102	8.67
Channel_Equalization	41	28	2101	3211	138238	138238	8.39
generateLLRs	1	0	65	143	4105	4105	4.10
hardDecodeLLRs	0	0	53	89	7685	7685	4.10
serialtoparallel	0	0	115	198	5,125	5,125	4.10
Receiver	**91**	**50**	**4388**	**6096**	**200258**	**200259**	**8.67**

Table 4.3: Timing and area information of unoptimized receiver

tion detects preamble using Schmidl-Cox algorithm in $input\_data$ and on detection it transfers the input data to $memstore$ which stores the incoming data and removes the cyclic prefixes, stores the pilots in $extracted\_pilots$ and data in $extracted\_data$. $extracted\_pilots$ has length of 128 and $extracted\_data$ has 4 blocks with 512 symbols in each. The $extracted\_pilots$ are fed to $Channel\_Estimation$ function which produces $est\_channel$.
The $Channel\_Estimation$ has a length of 20, which is zero appended to make of length 512. The $Channel\_Equalization$ function takes the $est\_channel$ and $extracted\_data$ to produce $equalized\_data$.
The $generateLLRs$ function takes the $equalized\_data$ which is phase corrected and produce LLR. The pilots inserted in the data are used for phase correction. The $LLRLoop$ takes data from the $DataLoc$ and puts real and imaginary parts in LLR. The $hardDecodeLLRs$ decode the bits and finally $serialtoparallel$ to $data\_out$, the message bits.

The top-level function of the receiver is $scfdma_rx$. The function prototype is as follows.

```
void scfdma_rx(
    myStream <data_t> input_data,
    ap_uint<16>  data_out[160]);
```

Basic optimizations on receiver Algorithm which was implemented were synthesized by applying basic transformations namely dataflow pipelining. The table 4.4 shows the result before and after dataflow pipelining of the receiver. Clearly this result is very far away from a real-time implementation since the initiation interval (II) is extremely large compared to our target of 12,000 clock cycles.

In order to analyze further we see the component wise breakup of the receiver as shown in the table 4.3.

The main optimization objective in receiver design is to minimize the Iteration Interval (II) which maximizes the throughput of the design. The minimizing iteration interval design strategy

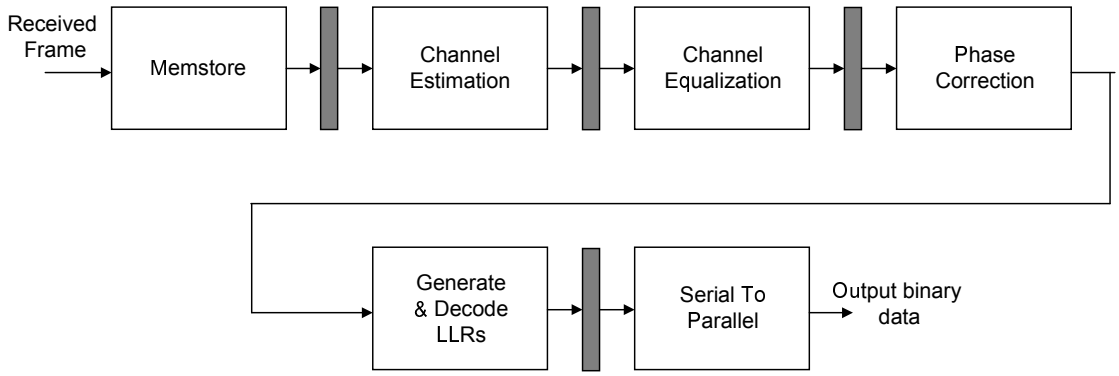

Figure 4.2: Receiver after dataflow pipelining

	BRAMs	**DSP48Es**	**FFs**	**LUTs**	T_L	**II**	T_{clk} **(ns)**
Before dataflow	91	50	4388	6096	200258	200259	8.67
After dataflow	132	50	4489	5871	74399	57482	8.67

Table 4.4: Receiver results before and after dataflow pipelining

(**?**) is to process as much as data as possible in the fewest cycles. The core idea is to apply data flow at function level and loop pipelining to each loop. In the receiver design, the top function is data-flow pipelined. The Figure 4.2 shows the data-flow pipelined design for receiver. After data flow pipelining additional 41 BRAMs were used. These BRAMs were used to store the data at intermediate stages and helps to reduce II. It can also be seen that the II is determined largely by channel equalization and channel estimation which are having II of 57482 and 14850 respectively even though all other modules are well within the design constraint of 12000 cycles.

It is clear that a simple synthesis of the receiver will not be sufficient to meet the required throughput constraints. We now describe each of the sub-functions in the order they are called, and study the optimizations applied at each stage to improve the resulting architecture.

4.2.1 Channel Estimation

Algorithm 5 DFT based Channel estimation algorithm

1: **procedure** CHANNEL_ESTIMATION($extracted\_pilots, est\_channel$)

Input: $extracted\_pilots$ are received pilots for channel estimation

Output: $est\_channel$ is the output of channel estimation algorithm used for equalization

2: $direction \leftarrow true$
3: FFT($extracted\_pilots, pilots\_CE\_freq, direction$)
4: EnvelopLoop:
5: **for** $i = 1 \rightarrow NO\_CE\_PILOTS$ **do**
6: $Envelop[i] \leftarrow pilots\_CE\_freq[i] \times PilotsFreq^{\star}[i]$
7: **end for**
8: MatIFFTLoop:
9: **for** $i = 1 \rightarrow CHANNEL\_LENGTH$ **do**
10: **for** $j = 1 \rightarrow NO\_CE\_PILOTS$ **do**
11: $est\_channel[i] \leftarrow$
$ChEstMat[i * NO\_CE\_PILOTS + j] \times Envelop[i]$
12: **end for**
13: **end for**
14: **end procedure**

The algorithm 5 describes the function $Channel\_Estimation$. In order to obtain the $est\_channel$ the received $extracted\_pilots$ are correlated with the known pilots. The correlation is performed in frequency domain by using DFT. If $Y(z)$ is the frequency response of $extracted\_pilots$ and $X(z)$ is the frequency response of the transmitted pilots then the estimated channel is calculated as $\hat{H}(z) = Y(z)/X(z)$. In the discrete domain this is approximated as $\hat{H}(n) = Y(n)/X(n)$. The FFT function computes 128-point FFT of the $extracted\_pilots$. The $pilots\_CE\_freq$ is multiplied with the complex conjugate of the $PilotsFreq$ vector to obtain the $Envelop$. The $MatIFFTLoop$ multiplies the $Envelop$ with the $ChEstMat$, which is an Inverse-DFT matrix to produce the $est\_channel$. The $est\_channel$ has most of the power concentrated in the first few coefficients as symbol period is longer than the duration of the channel impulse response (Kang *et al.* (2007)). The channel impulse response is limited to length of 20 as mostly noise exists outside the first few coefficients. The inverse-DFT for first 20 values is computed using matrix multiplication.

	BRAMs	DSP48Es	FFs	LUTs	T_L	II	T_{clk} (ns)
Before Optimizations	36	22	1938	2093	26656	14850	8.67
After Optimizations	67	96	7319	5291	12348	11807	8.33

Table 4.5: Channel estimation results before and after optimizations

Optimizations

As per the iteration interval minimization strategy, loops contained in the functions are pipelined. The vector-vector multiplication in the $EnvelopLoop$ is pipelined. The inner $MatIFFTLoop$ of the matrix-vector multiplication is pipelined which is used to maximize the throughput. This is accomplished using the pragma

```
#pragma HLS PIPELINE
```

The FFT is used both in the channel estimation as well as equalization phases. In the next section we will see some of the optimizations that are tried out in FFT.

4.2.2 FFT

The FFT used in the system is described in Salaskar (2016) which uses the burst mode of operation of the FFT. The same set of optimizations has been tried out to check whether the hardware synthesis results holds good with the new hardware resource constraints.

Algorithm 6 describes the standard Decimation In Frequency Fast Fourier Transform (DIF-FFT) algorithm. This is a 2-loop iterative algorithm modified from the standard 3-loop algorithm (Liu *et al.* (2009)). The outer loop counts the number of stages and inner loop count over the number of butterfly computations. The number of stages in the N-point FFT computations is $\log_2 N$ and number of butterfly computations in a stage is $\frac{N}{2}$.

The $index$ variable provides the index of the butterfly coefficients and $bottom$ and top variables provide the lower and upper addresses in Xin array. The dir variable decides whether the operation to be performed is FFT or Inverse-FFT. This algorithm performs in-place computation, so the variable Xn is read and written in a loop.

The Procedure $ComputeButterfly$ describes the butterfly operation. The additions and multiplications are performed on complex numbers. The dir variable chooses between forward

and inverse FFT (complex conjugate of the twiddle factor). The FFT/IFFT output is scaled by 2 at each stage similar to Xilinx FFT IP. The twiddle factor values are pre-computed and stored in the ROM.

Algorithm 6 DIF-FFT algorithm

procedure FFT(Xin, Xk, dir)
  for $i = 1 \rightarrow N$ **do**
    $Xn(i) \leftarrow Xin(i)$ ▷ buffer incoming data
  end for
  for $Stages = 1 \rightarrow \log_2(N)$ **do** ▷ Running over number of stages
    for $Butterfly = 1 \rightarrow \frac{N}{2}$ **do** ▷ Butterfly in a stage
      $index \leftarrow butterflyCoefficientIndex$
      $bottom \leftarrow LowerAdressInXn$
      $top \leftarrow UpperAdressInXn$
      COMPUTEBUTTERFLY($Xn, index, top, bottom$)
    end for
  end for
  BITREV(Xn, Xk) ▷ Arrange data in normal order
end procedure
procedure COMPUTEBUTTERFLY($Xn, index, top, bottom, dir$)
  if dir **then**
    $Xn[top] \leftarrow (Xn[top] + Xn[bottom])$
    $Xn[bottom] \leftarrow (Xn[top] - Xn[bottom]) \star Twiddle[index]$
  else
    $Xn[top] \leftarrow (Xn[top] + Xn[bottom])$
    $Xn[bottom] \leftarrow (Xn[top] - Xn[bottom]) \star Twiddle[index]^{\star}$
  end if
end procedure
procedure BITREV(Xn, Xk)
  for $Stages = 1 \rightarrow N$ **do**
    $Xk[BitRevAddr[i]] \leftarrow Xn[i]$
  end for
end procedure

Optimizations

A number of optimizations similar in lines with Salaskar (2016) were tried out and achieved a good implementation of the FFT.

Unoptimized The estimated hardware usage of the unoptimized FFT implementation is given in table 4.6.

BRAM_18K	DSP48E	FF	LUT	T_L	II	T_{clk} (ns)
5	6	461	668	11806	11806	8.33

Table 4.6: Resource usage and latency for unoptimized design

It is clear from the results obtained for unoptimized design that the default implementation by Vivado HLS minimizes resource consumption and the level of parallelism. The design uses 5 BRAM_18Ks. It uses 2 BRAMs for storing sine and cosine values i.e. twiddle factors. It uses another 2 BRAMs for storing the buffered input complex values (one each for real and imaginary part) and one for storing the bit reverse addresses.

The design uses 5 DSP48E (Xilinx hard macro for multiply/add type operations) for complex multiplications involved in butterfly computation and one for computing the index for coefficients, thus utilizing total of 6 DSP48E blocks.

As the iteration interval is one more than the latency we can see that no hardware parallelism has been exploited in the design. The optimizations that are performed in the FFT are the following

PipelinedLoops In order to reduce latency of the design we **pipeline** all the three loops. The results after pipelining are in table 4.7. From these results, we see that, though the design uses

BRAM_18K	DSP48E	FF	LUT	T_L	II	T_{clk} (ns)
5	6	530	803	7942	7942	8.33

Table 4.7: Resource usage and latency after Pipelining Loops

same number of BRAM_18K and DSP48E as unoptimized design, the latency/interval has decreased and the clock period achieved has increased. The design is not pipelined in the butterfly computation because of carried dependency. This is a dependency between operations in an iteration of a loop and an operation in a different iteration of the same loop. The tool by default assumes that a read cannot be performed before the write from the previous iteration is complete.

DependenceRemoval In order to remove dependency on variable Xn we specify inter iteration dependence removal directive. After applying this optimization, results are in Table 4.8

BRAM_18K	DSP48E	FF	LUT	T_L	II	T_{clk} (ns)
5	6	563	804	5640	5640	8.33

Table 4.8: Resource usage and latency after dependence Removal

From these results, we see that the design uses same number of BRAM_18K and DSP48E, the latency/interval has decreased and the clock period achieved has decreased.

BitRev The $BitRev$ function is shown in algorithm 25 uses pre-computed bit reversed addresses to perform the bit reverse.

The resulting FFT architecture meets the throughput requirements as well as being quite compact. The FFT is used in multiple places in the channel equalization section of the communication system. The compact nature of the block allows us to instantiate multiple instances of FFT.

4.2.3 Channel Equalization

The algorithm 7 describes the function $Channel\_Equalization$. The $est\_channel$ is appending with zeros to make it equal length as symbol of $extracted\_data$. The equalization is performed in frequency domain. The FFT of $est\_channel$ is computed as $EstimatedChannel\ Freq$. The $FFTLoop$ takes 512-point FFT for each symbol of $extracted\_data$ to convert into frequency domain as $RxDataFreq$. The $EqualizeLoop$ multiples each symbol of $RxData\ Freq$ with the complex conjugate of $EstimatedChannelFreq$ and divides by the square magnitude of the $EstimatedChannelFreq$. The $IFFTLoop$ takes back the $EqDataFreq$ data to time domain as $equalized\_data$.

Optimizations

Data flow pipelining has been tried in the function $Channel\_Equalization$ which allows FFT, equalization and IFFT operations to operate in parallel. The $extracted\_data$, $RxDataFreq$, $EqDataFreq$ and $equalized\_data$ are partitioned into 4 blocks of 512 symbols, which allows all the FFT and IFFTs to operate in parallel. The $IFFTLoop$ is unrolled. The $EqualizeLoop$

Algorithm 7 Frequency Domain Channel equalization

1: **procedure** CHANNEL_EQUALIZATION(extracted_data,est_channel,equalized_data)

Input: $extracted\_data$ received data for equalization,$est\_channel$ estimated channel for equalization

Output: $equalized\_data$ output data after equalization

2: Append zeros to make $est\_channel$ length equal to $BLOCK\_LENGTH$
3: $direction \leftarrow true$ ▷ select mode as FFT
4: ▷ FFT of estimated channel
5: FFT($est\_channel, EstimatedChannelFreq, direction$)
6: FFTLoop: ▷ Data to frequency domain
7: **for** $i = 1 \rightarrow NO\_BLOCKS$ **do**
8: FFT($extracted\_data, RxDataFreq, direction$)
9: **end for**
10: EqualizeLoop: ▷ Equalize in frequency domain
11: **for** $i = 1 \rightarrow NO\_BLOCKS$ **do**
12: **for** $j = 1 \rightarrow BLOCK\_LENGTH$ **do**
13: $EqDataFreq[i][j] \leftarrow RxDataFreq[j] \times \frac{EstimatedChannelFreq^*[j]}{|EstimatedChannelFreq[j]|^2}$
14: **end for**
15: **end for**
16: $direction \leftarrow false$ ▷ select mode as IFFT
17: IFFTLoop: ▷ Data back to time domain
18: **for** $i = 1 \rightarrow NO\_BLOCKS$ **do**
19: FFT($EqDataFreq, equalized\_data, direction$)
20: **end for**
21: **end procedure**

is pipelined to reduce the iteration interval. The Figure 4.3 shows the result after application of all the optimizations.

4.2.4 Phase Correction

The algorithm 8 describes the function $getCordicAndCorrect$. The $ExtractPilotsLoop$ nested loop extracts pilots from the symbols from the $extracted_data$ in to the $RxPilots$ using $Mask$. The $AnglePilotsLoop$ converts the $RxPilots$ complex values from (x, y) to (r, θ) using Vectoring mode of $Cordic$ function and accumulates the angle for all the extracted pilots in a symbol. The $AvgAngle$ is calculated by dividing the accumulated angle by NO_PILOTS belonging to one block of symbols. The $AvgAngle$ with magnitude of one is converted from (r, θ) to (x, y) using rotation mode of $Cordic$ function to $PhaseErr$. The $equalized_data$ is

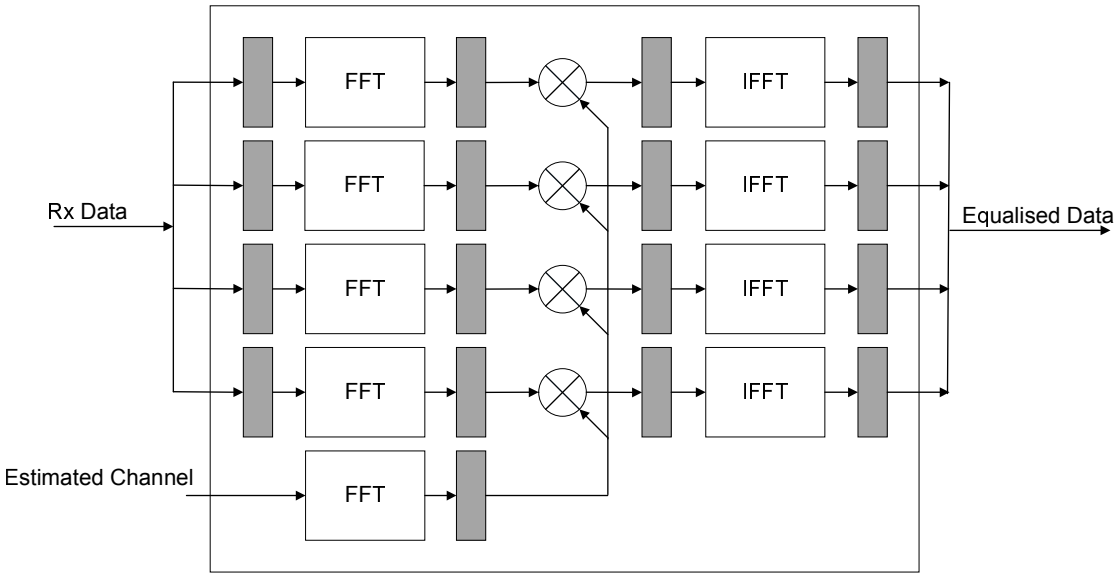

Figure 4.3: Block Diagram of Channel Equalization

	BRAMs	**DSP48Es**	**FFs**	**LUTs**	T_L	**II**	T_{clk} **(ns)**
Before Optimizations	41	28	2101	3211	138238	138238	8.39
After Optimizations	152	52	5259	7283	12888	12888	8.39

Table 4.9: Channel equalization results before and after optimizations

	BRAMs	**DSP48Es**	**FFs**	**LUTs**	T_L	**II**	T_{clk} **(ns)**
Before Optimizations	3	12	867	1289	4293	4293	8.39
After Optimizations	3	12	1962	5327	1322	1322	8.65

Table 4.10: Phase correction results before and after optimizations

multiplied by the complex conjugate of the $PhaseErr$ to get $PhCorrectData$.

Optimizations

The unoptimized phase correction module does not use any BRAMs for pilots locations or pilot values. After pipelining the $ExtractPilotsLoop$, $AnglePilotsLoop$ and $PhaseCorrLoop$, the pilots locations and pilots values can be stored in BRAMs. This causes decrease in number of FFs and LUTs but increase in BRAMs and $4\times$ decrease in the iteration interval.

The algorithm 9 describes the CORDIC algorithm. There are two modes: vectoring mode and rotation mode. It is chosen using the $Mode$ variable. If the mode is rotation mode then x

Algorithm 8 Phase Correction

1: **procedure** GETCORDICANDCORRECT($equalized\_data, PhCorrectData$)

Input: $equalized\_ata$ input data for Phase correction

Output: $PhCorrectData$ output data after phase correction

2: ExtractPilotsLoop: ▷ Extract pilots from symbols
3: **for** $i = 1 \rightarrow NO\_BLOCKS$ **do**
4: **for** $j = 1 \rightarrow BLOCK\_LENGTH$ **do**
5: **if** $Mask[j] = PILOT$ **then**
6: $RxPilots[m] \leftarrow equalized\_data[i][j]$
7: $m \leftarrow m + 1$
8: **end if**
9: **end for**
10: $Angle \leftarrow 0$
11: $Mode \leftarrow false$ ▷ Select Vectoring mode
12: AnglePilotsLoop:
13: **for** $i = 1 \rightarrow NO\_PILOTS$ **do**
14: $AngleVector \leftarrow RxPilots[i] \times Pilots^{\star}[i]$
15: CORDIC(AngleVector.Real,AngleVector.Imag,Theta,Mag,Mode)
16: $Angle \leftarrow Angle + Theta$ ▷ Accumulate angle over all pilots
17: **end for**
18: $AvgAngle \leftarrow \frac{Angle}{NO\_PILOTS}$
19: $Mode \leftarrow true$ ▷ Select Rotation mode
20: CORDIC(Phase.Real,Phase.Imag,AvgAngle,Mag,Mode)
21: PhaseCorrLoop:
22: **for** $i = 1 \rightarrow NO\_BLOCKS$ **do** ▷ Apply phase correction
23: $PhCorrectData \leftarrow equalized\_data \times PhaseErr^{\star}$
24: **end for**
25: **end for**
26: **end procedure**

is initialized with the constant multiplication factor, y as 0 and z to the angle to be rotated θ as described in the section 3.2.6. Similarly in vectoring mode, (x, y) is assigned input vector $(x0, y0)$ and z as zero. The CORDIC iterations are performed $CORDIC\_TAB$ number of times. In each iteration (x, y, z) tuple is updated based on vectoring or rotation mode. In rotation mode the update is made based on value of z while in vectoring mode update is based on y. The new value of x and y is calculated based on previous values of x and y and the iteration count. The z is added/subtracted values from $CORDIC\_TAB$ which are pre-computed values of $\tan^{-1}$.

The unoptimized CORDIC module has iteration interval of 33, while after pipelining the

Algorithm 9 Cordic algorithm

procedure CORDIC($x0, y0, \theta, r, Mode$)
  if $Mode = true$ **then** ▷ Rotation Mode
    $x \leftarrow CORDIC\_1K, y \leftarrow 0, z \leftarrow \theta$ ▷ Init with scaling factor and angle to be rotated
  else ▷ Vectoring Mode
    $x \leftarrow x0, y \leftarrow y0, z \leftarrow 0$ ▷ Init vector
  end if
  for $i = 1 \rightarrow CORDIC\_NTAB$ **do**
    if $Mode = true$ **then** ▷ Rotation Mode
      if $z < 0$ **then**
        $x_t \leftarrow x + y >> i$
        $y_t \leftarrow y - x >> i$
        $z_t \leftarrow z + CORDIC\_TAB[i]$
      else
        $x_t \leftarrow x - y >> i$
        $y_t \leftarrow y + x >> i$
        $z_t \leftarrow z - CORDIC\_TAB[i]$
      end if
    else ▷ Vectoring Mode
      if $y < 0$ **then**
        $x_t \leftarrow x - y >> i$
        $y_t \leftarrow y + x >> i$
        $z_t \leftarrow z - CORDIC\_TAB[i]$
      else
        $x_t \leftarrow x + y >> i$
        $y_t \leftarrow y - x >> i$
        $z_t \leftarrow z + CORDIC\_TAB[i]$
      end if
    end if
    $x \leftarrow x_t, y \leftarrow y_t, z \leftarrow z_t$ ▷ Update
  end for
  $x0 \leftarrow x, y0 \leftarrow y, \theta \leftarrow z, r \leftarrow x \times CORDIC\_1K$
end procedure

	BRAMs	DSP48Es	FFs	LUTs	T_L	II	T_{clk} (ns)
Before Optimizations	0	0	286	843	33	33	8.39
After Optimizations	0	0	574	4875	4	1	8.65

Table 4.11: CORDIC results before and after optimizations

CORDIC function, we are able to achieve iteration interval of 1 for CORDIC block. We get about 6× increase in LUTs and 2× increase in FFs while decreasing iteration interval from 33 to 1.

4.2.5 Discussion

This chapter reveals that the complexity of the receiver structure is more than the transmitter, and a direct C/C++ implementation results in a design which will not meet the throughput requirements. A major chunk of the optimizations involve loop pipelining and loop unrolling but in order to have full benefits of the optimizations modification in algorithm also is required. By applying all these directives the transceiver synthesis results shows that the initiation interval is kept below the desired.

In the next chapter we deal with the final consolidated transceiver results followed by details of testing on the actual hardware, FPGA board (XC7V585T-FFG1761) from DEAL, DRDO,Dehradun.

4.3 Integration using Vivado IP Integrator

The IP level block diagram of the SC-FDMA system integrated with various peripherals is shown in the Figure 4.6,where data flow for transmitter chain and receiver chains are shown in separate colours. We took a two way approach for achieving the objective of implementing the hardware on customized FPGA board. First implement the hardware and software platforms in VC707 evaluation board and then retarget the design for customized board as implementing and evaluating the design on an evaluation board will be easier compared to the latter. The whole block diagram is built around AD9364 RF transceiver IP's reference design. Required modifications are done on the block diagram by integrating SC-FDMA project related customized IP's. As the embedded software required to run on the reference design is LwIP echo server (custom modified), then minimal hardware required for that application are added along with

the reference design. They are

- Processor - used is Microblaze softcore processor.
- Ethernet MAC - used is Axi-Ethernet to send and receive packets.
- Interrupt Controller - used is Axi-Intc.LwIP adapters work in interrupt mode only.Hence the packet reception or transmission is notified to software via interrupts by the ethernet MAC. It also connects multiple source of interrupts to the processor.
- Programmable Timer - used is Axi-Timer. LwIP required periodic interrupts to update TCP timers.

The Figure 4.4 depicts the data flow which is shown by red colour in actual hardware during transmission without ethernet. Microblaze also configures the AD9364 module. The Figure 4.5 depicts the data flow in actual hardware during reception without ethernet and the Figure 4.6 depicts the block design of the integrated SC-FDMA system in Vivado IP Integrator.

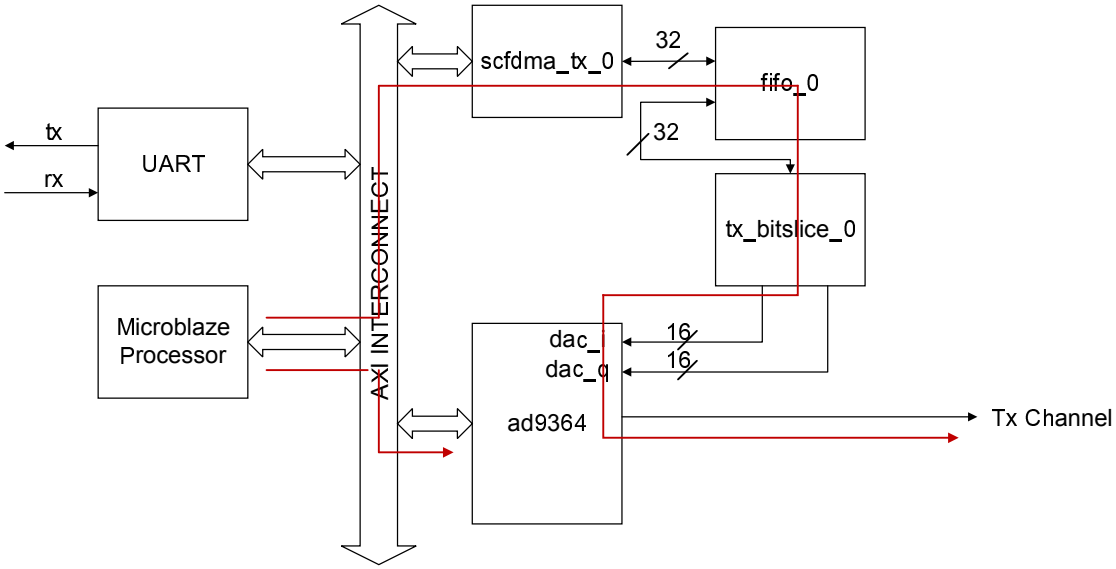

Figure 4.4: Integration of SC-FDMA transmitter system on Vivado IP Integrator

The **scfdma_tx_0** IP takes 160 words of 16 bit data as input and produces the 2400 symbols of 16 bit each of I and Q data. The data at the output of the **scfdma_tx_0** IP is 32 bit (I and Q combined). It is fed to a Xilinx FIFO **fifo_0** IP which is having the write (100MHz) and read (20 MHz) clock cycles of different frequency. This is because the receiver IP **scfdma_rx_0** is limited to 20 MHz as its overall latency is close to 12000 cycles. Hence reading from the FIFO will be done at a slower pace and is fed to a customized IP **tx_bitslice** which divides the 32 bit word into 16 bit I and Q samples. The I and Q samples is fed to DAC of RF transceiver IP AD9364 where it is called baseband signal and is upconverted to the transmitter LO of set frequency (configured through SDK) now called band pass signal and is combined and output as analog differential signal on the Tx Channel.

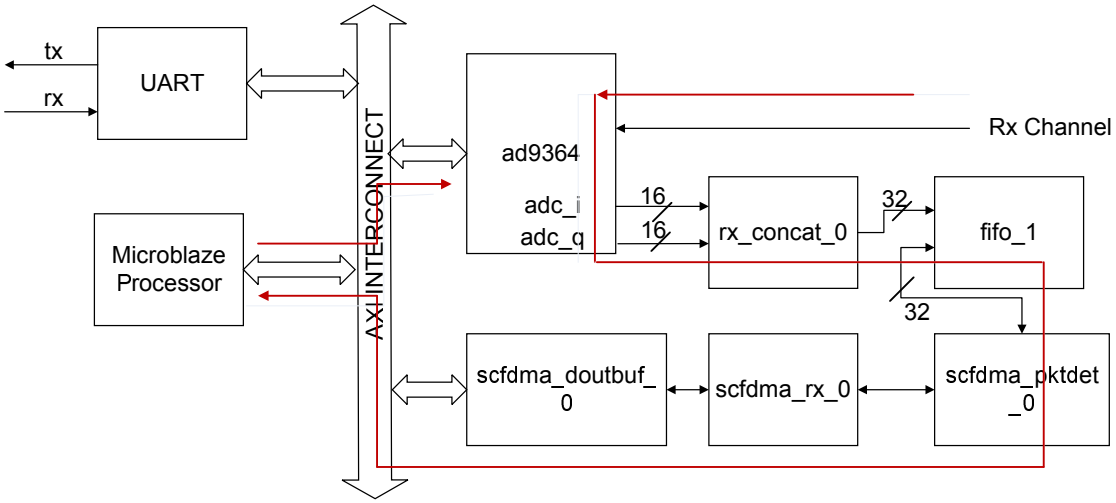

Figure 4.5: Integration of SC-FDMA receiver system on Vivado IP Integrator

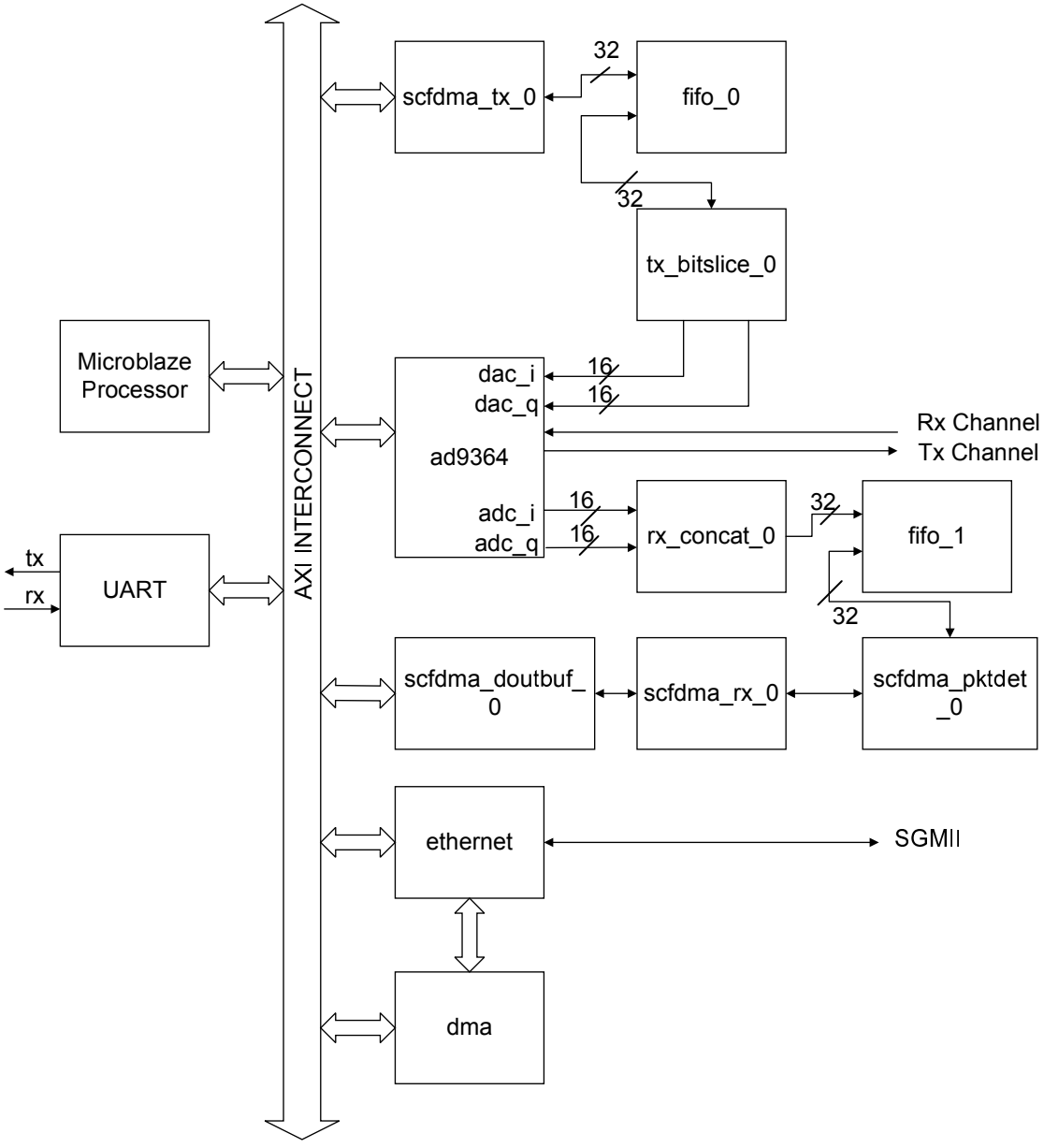

Figure 4.6: Integration of SC-FDMA system on Vivado IP Integrator

In testing using a single board, the Tx Channel and Rx Channel are looped back externally using a SMA cable. Hence the bandpass signal thus received through Rx Channel is down converted using the set receiver LO set frequency (configured through SDK - for single board both these frequencies must be same say 2.1 GHz or say 2.4 GHz) called base band signal and is converted from analog to digital and divided into its I and Q components. It is combined into a 32 bit word using **rx_concat**. RF transceiver contains 12 bit σ-δ ADC and DAC. So necessary logics has been implemented in **tx_bitslice** and **rx_concat** without compromising on the accuracy.
The combined digital data written to the FIFO IP **fifo_1** is fed to the **scfdma_pktdet_0** where the packet detection happens and the data is fed to the **scfdma_rx_0** where it is decoded and through the **scfdma_doutbuf_0** it is read by the processor using an ISR through the AXI bus.

4.4 Embedded Software using Xilinx SDK

The embedded software application running on the integrated system (Microblaze processor and customised SC-FDMA transmitter and receiver IP's) is based on the LwIP echo server application. A minimum requirement of 2MB space of RAM is required for the application to run. Firstly ,a board support package (BSP) has been created which is a collection of libraries and drivers that will form the lowest layer of the application software stack. The embedded software application must run on top of a the software platform using the APIs that it provides. In this project we are using standalone OS for creating BSP. When the BSP is built, it includes standard C libraries and device drivers for all the peripherals that are used in the project.The customised IP's **scfdma_tx_0** and **scfdma_doutbuf_0** are *AXI-Lite* based whereas **scfdma_pktdet_0** and **scfdma_rx_0** is of type *ap_ctrl*. IP's which are using AXI-Lite interfaces created its own API's as part of Vivado-HLS implementation itself and is made available at the SDK library through the export hardware wizard of Vivado IP Integrator.
The Figure 4.7 shows the flow chart of the initialization routines, Figure 4.8 shows the flow chart showing the events happening during transmission and the Figure 4.9 shows the events happening during the reception of SC-FDMA with modified LwIP echo server application which has been used to test the integrated system. The function call associated with each event also is given side by side to the flow chart for easy refernce. The orange colour blocks on the flow chart shows the events that are happening on the hardware and the rest describes the events that are driven by software.

In Xilinx SDK create a new application project using the LwIP echo server application which uses RAW APIs with standalone OS and hardware platform exported from Vivado IP

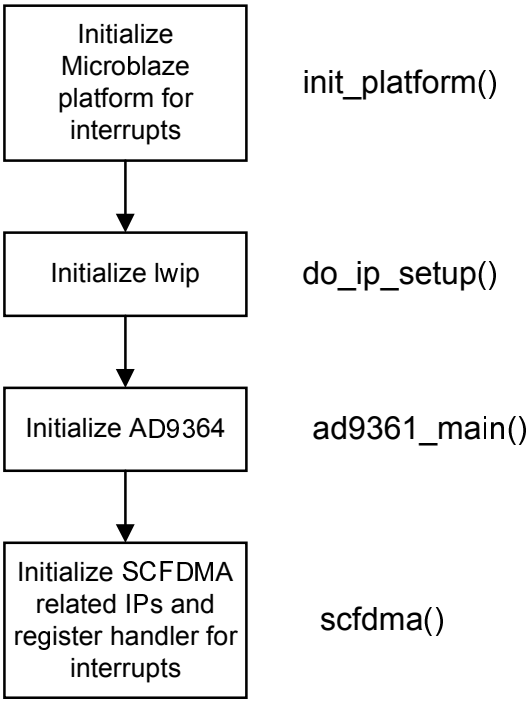

Figure 4.7: Flow chart of SCFDMA embedded software initialization routine

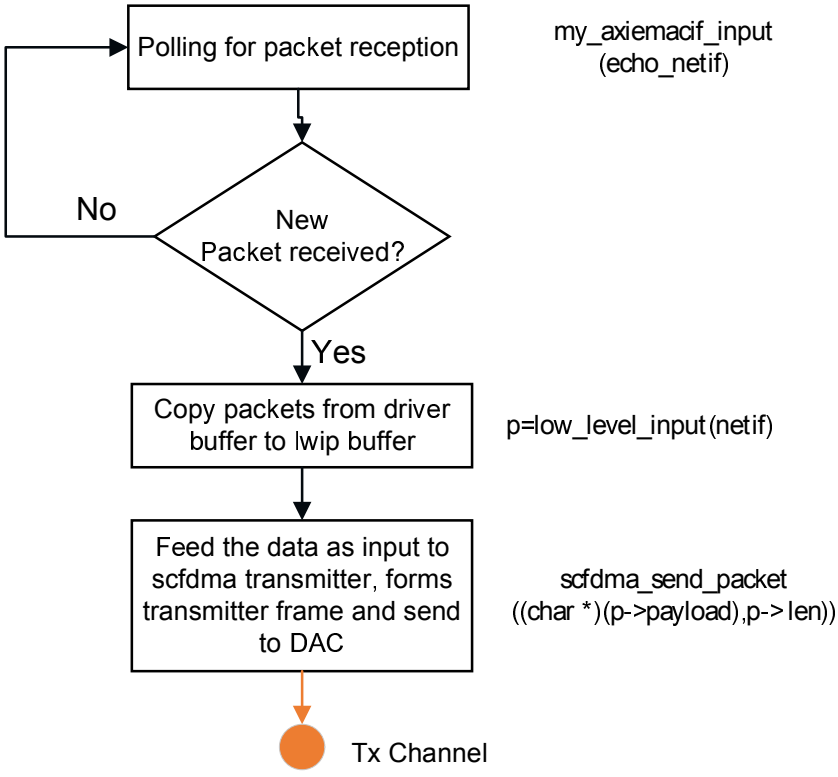

Figure 4.8: Flow chart of SCFDMA embedded software transmitter routine

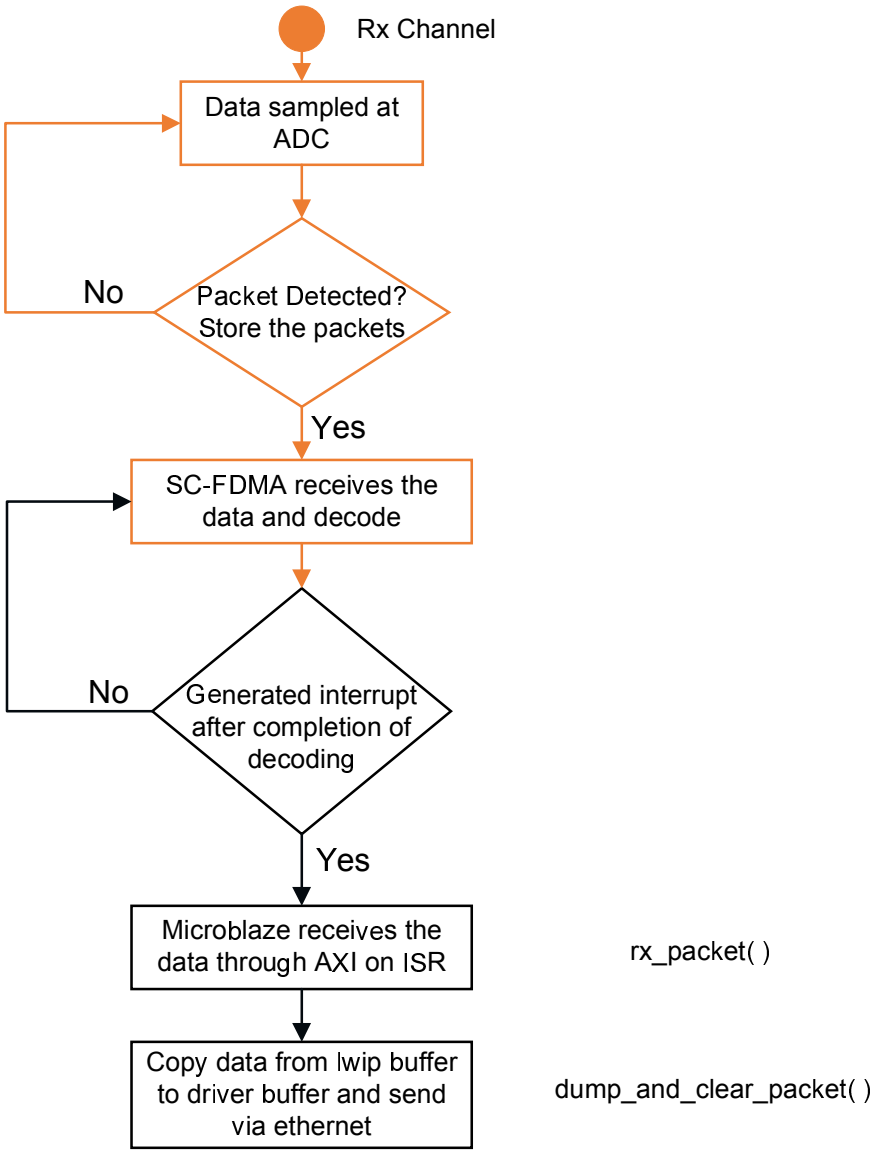

Figure 4.9: Flow chart of SCFDMA embedded software transmitter routine

integrator. The following changes needs to be done on the application side. Add the following line

```
options |= XAE_PROMISC_OPTION;
```

to the function

```
void init_axiemac(xaxiemacif_s *xaxiemac, struct netif *netif)
```

in the file

```
bsp/sys_mb/libsrc/lwip141_v1_5/src/contrib/ports/xilinx/
netif/xaxiemacif_hw.c
```

and also add the following to the main.c file so that the IP can be set manually.

```
#undef LWIP_DHCP
#undef LWIP_ARP
```

The XAE_PROMISC_OPTION is set to receive all packets that are appearing on the network interface at ethernet level. After all initialization routines and manual setting of IP's are over the application waits in polling mode to check if a packet is received. When a packet is received, it is first copied from the ethernet driver buffers into the LwIP buffers using DMA in SG mode. In order to copy the packet as fast as possible, the LwIP buffers (pbufs) should be allocated from the pool of buffers (PBUF_POOL). When a packet has been copied, it will act as a payload for the **scfdma_tx_0** IP to take as input and form the transmitted frame and send via the RF transceiver and free the lwip buffer. As the ADC and DAC of RF transceiver is in loop back through physical connection using SMA cables, the packet is fed to **scfdma_pktdet_0**. Once the packet is detected the data is further received by **scfdma_rx_0** and the decoded data is received by the lwip buffer and is copied to the ethernet buffer using DMA in SG mode. So the data transfer from and into the ethernet interface is facilitated by DMA. The interface used for connecting MAC with PHY of the ethernet chip is through SGMII.

CHAPTER 5

Integration Testing and Results

This chapter provides the methods used to test the design on board using software running on the processor and debug the design using ILA. The transmitter and receiver are implemented in hardware using the Vivado^{TM}HLS tool from Xilinx. The VC707 evaluation board using the Virtex 7 FPGA from Xilinx was used in the initial implementation and later used customized FPGA (XC7V585T-FFG1761C) board from DEAL,DRDO. The table 5.1 gives the utilization and performance estimate for the transmitter and receiver. From the table 5.1 it is clear that Channel Estimation and Channel Equalization are the bottle-neck in order to improve the iteration interval of the receiver. As our timing constraint is met in terms of number of cycles consumed it is an accepted implementation.

The table 5.2 gives the post implementation result of the integrated SC-FDMA system The utilization of the complete integrated transceiver is well within the FPGA resources.The next section deals with the method of testing design on board.

5.1 Method of on-board testing and results

The first step towards validating the design has been completed in the Vivado-HLS tool through the various design flows namely C-Simulation,Synthesis,C/RTL cosimulation. However, the final verification of the system requires actual implementation in hardware.The customized board from DEAL has the provision to test the integrated system upto bandpass signal level. For testing purpose two hardware boards has been connected in a criss-cross fashion means the transmitter of one board is connected to the receiver of another board and vice-versa.The Figure 5.1 depicts the test setup.The boards are connected to the individual PC's through ethernet cable.The two FPGA boards physically connected with each other using SMA cables implemented with SC-FDMA system acts like a hub. The transmitter and receiver ports on the board are interfaced through SMA cables with two attenuators of 10dB,50 Ohms each connected in series with a single transmitter receiver pair. An external attenuation of 20 dB is provided inorder to bring the transmit channel power of -17 dB (without transmitting attenuation set in SDK) well within the lower and upper levels of input power handling capacity of -50 dB and -20 dB respectively of RF transceiver module for faithful reception of the signal. The Table 5.3 gives the configuration

Module	BRAMs	DSP48Es	FFs	LUTs	T_L	II	T_{clk} (ns)
WimaxEncoder_Loop1	7	0	674	2303	6501	6501	7.38
WimaxEncoder_Loop2	3	0	407	1200	2419	2419	4.81
Transmitter	**20**	**0**	**1199**	**3618**	**8921**	**6502**	**7.38**
Pktdet	0	20	2600	414	4	1	8.51
MemStore	0	0	92	84	2322	2322	4.38
ChannelEstimation	59	96	6496	6000	5649	5641	8.33
ChannelEqualization	99	64	9408	16423	1	6153	8.75
GetCordicAndCorrect	3	12	2067	5712	1323	1323	8.65
hardDecodeLLRs	0	0	48	158	3842	3842	5.39
SerialToParallel	0	0	116	167	2562	2562	6.52
Receiver	**213**	**160**	**17792**	**25882**	**8514**	**6153**	**8.75**
Virtex 7	**1590**	**1260**	**728400**	**364200**			

Table 5.1: Timing & area information for transmitter and receiver

Module	LUT	LUTRAM	FF	BRAM	DSP
SC-FDMA	82689	7882	118230	321.5	486
FPGA	364200	111000	728400	795	1260

Table 5.2: Post implementation result of SC-FDMA on DEAL FPGA board

parameters set for the RF transceiver

Parameter	Board 1 Value	Board 2 Value
Tx LO Frequency	2100000000	1900000000
Rx LO Frequency	1900000000	2100000000
Tx RF Bandwidth	35000000	35000000
Rx RF Bandwidth	35000000	35000000
Sampling Frequency	20000000	20000000
Tx Attenuation	10000	10000
Rx RF Gain	35	35

Table 5.3: Configuration settings for AD9364 RF Transceiver

5.1.1 Procedure to set up integration testing

- The boards and PC's has to be connected as shown in Figure 5.1.
- Communication between the boards should be through SMA cables with 20 dB attenuators and through ethernet cable with board and PC.
- Power ON the entire system.
- Start hyperterminal application like gtkterm on both PC's with baud rate set to 115200. (It should be same as baud rate set for UART through IP configuration wizard in Vivado IP integrator)

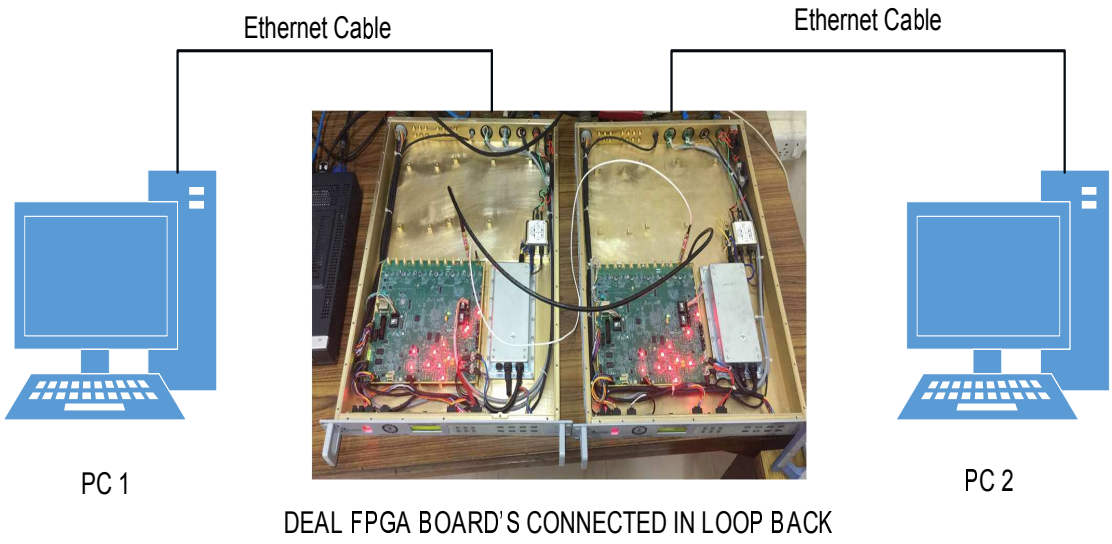

Figure 5.1: Diagram showing the testing of SC-FDMA system on actual hardware

- Set the IP of PC 1 and PC 2 using the command's as shown below

  ```
  ifconfig eth0 192.168.1.20 netmask 255.255.255.0 mtu 1400
  ifconfig eth0 192.168.1.30 netmask 255.255.255.0 mtu 1400
  ```

  reason for mtu set to 1400 is explained in section Integration issues and solutions.

- Download the bit file and application elf using JTAG and Vivado tool.
- Set the Tx and Rx LO frequencies as per the Table 5.3 through the UART user interface.
- After successful completion of all the initialization routines, take terminal on both PCs and try to **ping** the other PC.
- If reply for the **ping** command appears on the terminal, then it is sure that the whole system is working fine.

The **ping** command is used for testing the system . It checks the working of LwIP echo server application as well as SC-FDMA transmitter and receiver system. A typical message on the hyper-terminal is as shown below where TC denotes SC-FDMA Transmit Count, RC denotes Receive Count of SC-FDMA and EC denotes Error Count, where packets are not received properly.

5.1.2 Description of events with ping on the system SC-FDMA

After setting up the PCs with IPs as discussed above, when the user runs the **ping** command on one PC say PC1,the command is received by the FPGA board connected to its network. The ethernet driver buffer of the FPGA copies data to LwIP buffer which is used by the SC-FDMA transmitter as a payload for its transmision . The SC-FDMA carrier modulated signal is received by the receiver of the other FPGA board through the SMA cable and is decoded correctly by the

receiver and is copied to LwIP buffer. Finally the data is fed into the ethernet layer of the other FPGA board and is received by the PC2.Once the PC2 sees the message that there is a ping request from PC1 (if the message is encoded and decoded correctly by SC-FDMA) the PC2 will reply for this protocol. The **ping** reply goes to the transmitter of SC-FDMA of the other FPGA and the whole process once again repeats to reach PC1 as reply to its issued **ping** command. This way if the ping command gets reply then we can ensure that the SC-FDMA system is operating as intended. A typical debug prints on the hyperterminal gtkterm is as shown below.

```
Start PHY autonegotiation
Waiting for PHY to  complete autonegotiation
Autonegotiation complete
Waiting for Link to be up; Polling for SGMII core Reg
auto-negotiated link speed: 100
 Press '1 or 0' to configure Tx and Rx as follows
 '1' will set Tx = 2.1 GHz and Rx = 1.9 GHz
 '0' will set Tx = 1.9 GHz and Rx = 2.1 GHz+++ prod id: A
+++ registered clocks
+++ adc init
+++ init gain_tables
+++ setup complete
Set Tx LO Freq is : 1900000000
Set Rx LO Freq is : 2100000000
Press a key to continue:
Done.
Initialized TX and doutbuf
 TC 2 RC 0 EC 2
 TC 5 RC 3 EC 2
 TC 9 RC 7 EC 2
```

The response to ping command is also shown below

```
test@test4:~$ ping 192.168.1.20
PING 192.168.1.20 (192.168.1.20) 56(84) bytes of data.
64 bytes from 192.168.1.20: icmp_seq=1 ttl=64 time=1.44 ms
64 bytes from 192.168.1.20: icmp_seq=2 ttl=64 time=1.39 ms
```

5.1.3 Integration issues and solutions

CFO variations

Checksum was introduced in the application for the purpose of ascertaining the correctness of transmitted and received data and found that they were not matched properly indicating errors in decoded messages. As the loop back between transmitter and receiver of the same board worked correctly the issue was associated to drift in crystal oscillator frequencies. The same has been revealed through the analysis using a spectrum analyzer. SC-FDMA is very sensitive to clock variations and even should match to parts per billion(ppb). One of the probable solution was to use an external clock source and feed to both boards. But the internal connection between the SMA clock connector to the RF transceiver chip was intentionally left open on the FMC board and hence could not do experiments on the VC707 board.

Similar issue was repeated in the customized FPGA board and there was a potentiometer associated with tuning of the clock of the crystal and the issue was rectified.

MTU set to 1400

The SC-FDMA system takes input of 2560 bits at a time or 320 bytes and append 8 bytes including a string "iitm00" and length field as its header and sends it. Similarly at the receiver when the packet first received at the lwip buffer the 8 bytes will be stripped off before moving it to receiver for further processing. The observation was that the packet started dropping at the larger sizes of **ping** command and analysis shown that the dropping of packets was due to the packet size growing larger than the 1486 bytes as additional 14 bytes added by ethernet makes it to more than 1500 bytes and packets get dropped at the ethernet layer. Hence the solution was to reduce MTU to size less than 1486 and we chose 1400.

Buffer allocation failure

The processor creates buffer for the DMA in RAM and asks the DMA to transfer data through the ethernet by making use of these buffers. The problem observed after some time of data transactions. Error message of "unable to allocate pbuf in recv_handler" was raised. Analysis shown that the pbuf after used by the SC-FDMA transmitter was not getting released. The pbuf_free() function was added after calling scfdma_send_packet() in scfdma.c file and the problem was solved.

DMA failed to transfer using iperf in UDP

iperf using UDP sends a burst of data and the iperf was not showing throughput. Intuitively assumed that the problem might be due to the failure of DMA in handling burst data transactions and the read and write data bursts were increased from 16 to 256 in the configuration of DMA using Vivado IP integrator configuration wizard.

Flashing of Clock synthesizer to take internal clock

The wrapper code for the block design was provided by DEAL and during integration with the customized FPGA board on VIvado 2016.2 version it was found that the flashing of Clock Synthesizer was not proper resulting in the wrong generation of clock signal for the entire system. Checking on the clock signal test point on the board using Oscilloscope revealed that clock signal levels are not showing the CMOS voltage levels and there is a large variation in frequency instead of 40 MHz. Hence it was decided to comment the clock synthesizer portion on the system wrapper file and instead have a separate bit file to flash the clock synthesizer and it needs to be done only once.Bit file name is **synthAD9520_3.bit**. Clock synthesizer output is coming at J17 SMA connector on the board and always shows 40 MHz.

Flashing of clock synthesizer to take external clock source

In-order to confirm that the CFO variations causes the wrong decoding of data, there was a need to use the external clock source as reference clock and a separate flash file for clock synthesizer has been made for this purpose.Bit file is **synthAD9520_3_tcxo.bit** which can take only 10 MHz clock source as input. It should be applied through J15 SMA connector on the board and the signal shall be 1V Sine wave of 10 MHz.

Reset not working properly

There was a reset issue with the DEAL board which will ask for a power cycle each time even if we do some modification in SDK.The issue was identified and rectified as follows

In ad9361_main.c file, changed the value of GPIO_RESET_PIN_0 (parameters.h) from 2 to 34.

In platform.c file updated the following functions to that are mentioned below. This is required as the GPIO pin we are using > 32.

Change the following function

void gpio_direction(uint8_t pin, uint8_t direction)

ifdef_XPARAMETERS_PS_H_

XGpioPs_SetDirectionPin(gpio_instance, pin, direction);

XGpioPs_SetOutputEnablePin(gpio_instance, pin, 1);

else

uint32_t config = 0;

uint32_t tri_reg_addr;

if (pin >= 32)

tri_reg_addr = XGPIO_TRI2_OFFSET;

pin -= 32;

else

```
tri_reg_addr = XGPIO_TRI_OFFSET;
config = Xil_In32((gpio_config->BaseAddress +
tri_reg_addr));
if(direction)

config =  (1 « pin);

else

config |= (1 « pin);

Xil_Out32((gpio_config->BaseAddress + tri_reg_addr),
config);
endif
```

To

```
void gpio_data(uint8_t pin, uint8_t data)

ifdef_XPARAMETERS_PS_H_
XGpioPs_WritePin(gpio_instance, pin, data);
else
uint32_t config = 0;
uint32_t data_reg_addr;
if (pin >= 32)
data_reg_addr = XGPIO_DATA2_OFFSET;
pin -= 32;
else
data_reg_addr = XGPIO_DATA_OFFSET;
config = Xil_In32((gpio_config->BaseAddress + data_reg_addr));
if(data)

config |= (1 « pin);

else
```

config = (1 « pin);

Xil_Out32((gpio_config->BaseAddress + data_reg_addr), config);
endif

Noisy data on ADC Channel

ADC output was observed noisy for single tone frequency. The problem was identified as a wrong connection in the block diagram design and done a connection change in block diagram by changing the DAC and ADC ENABLE signal going to FIFOs rd_en and wr_en to corresponding VALID signals.

Flashing the application to enable autoboot after poweroff or reset

Auto boot of application from flash is not able to do because it requires a windows utility to flash the NOR flash as SDK supports only BPI flash devices.

CHAPTER 6

Conclusions and Future Work

A complete end-to-end implementation of a specific wireless communication system using the Vivado-HLS, Vivado IP Integrator and Xilinx SDk tools from Xilinx has been presented. The system was coded using C++; simulated for correctness, and then synthesized to RTL. The final design was demonstrated to work on an FPGA platform.
Some of the observations and conclusions are as follows

- By default Vivado HLS optimise resources. It tends to have high latency and the level of parallelism achieved is minimum.
- Higher degree of parallelism and improved throughput can be achieved by judicious use of directives involving loop unrolling, pipelining, and memory partitioning.
- A large number of design alternatives can be explored as coding and testing through simulation are much faster at the C++ level than at RTL.

In summary, the HLS tool provides a convenient way to explore and synthesize a highly efficient implementation of the design, but requires a fairly high amount of manual intervention to achieve good results. In the present instance, suitable tuning of the code and directives allowed us to achieve performance at the cost of increasing hardware.

Future Work The major objective of the work was to implement the design on the FPGA board provided by DEAL and achieve the specified data rate. The future work can concentrate on how to improve the data rate further by the efficient use of HLS tools.
In order to increase the data rate significantly we need to increase parallelism in the design. The bottlenecks in the design are FFT blocks for which architectures like pipelined architectures can be explored to reduce the iteration interval. In all these cases these is a trade off between the hardware and iteration interval. In the present instance we have chosen a relatively simple architecture for the FFT that allows multiple FFT units to be instantiated in parallel in order to achieve the overall computation. In the present design we have not included the LDPC as it takes more latency than the desired data rate requirement. The future work can concentrate on the implementation of LDPC in the system. Another drawback of the present system is effect of channel is not added along with the transmitted data. Practical systems has to mitigate for

channel impairements and in order to have the facility for taking the channel effects into consideration LDPC or channel coding schemes has to be implemented.

The LwIP echo server application can be customised further to facilitate throughput measurement using tools like iperf. In the present system, the throughput measuremt using iperf did not show up convincing results. This can be explored further.

The transmitter and receiver modules can be further divided into unique functional blocks such that any complex systems can be build by using a combination of these blocks in a simple plug and play manner. Parameterized implementation in the design can be brought out to accommodate addition of more blocks of symbols or changing size of block to adapt for different system, support for different modulation schemes such as QAM and others, support for MIMO etc. The use of template types of C++ increase parameterization in the design.

REFERENCES

1. **Donald.G.Bailey**, The advantages and limitations of high level synthesis for fpga based image processing. *In International Conference on Distributed Smart Cameras*. 2015.

2. **Kang, Y.**, **K. Kim**, and **H. Park** (2007). Efficient DFT-based channel estimation for OFDM systems on multipath channels. *Communications, IET*, **1**(2), 197–202. ISSN 1751-8628.

3. **Liu, X.**, **X. Song**, and **Y. Wang**, Performance evaluation on FFT software implementation. *In Proceedings of the International MultiConference of Engineers and Computer Scientists*. 2009.

4. **Proakis, J. G.** and **D. G. Manolakis**, *Digital Processing 4th Edition*. Prentice Hall, New Jersey, 2006.

5. **Salaskar, A.** (2016). Implementation of a single carrier frequency domain equalization transceiver using high level synthesis.

6. **Schmidl, T. M.** and **D. C. Cox**, Robust frequency and timing synchronization for OFDM. *In IEEE Transactions on Communications*, volume 45, number 12. 1997.

7. **Volder, J. E.**, The CORDIC trigonometric computing technique. *In IRE Trans. Electron. Computers*, volume EC, number 8. 1959.

8. **Walther, J. S.**, A unified algorithm for elementary functions. *In Proc. 38th Spring Joint Computer Conf.*. 1971.

9. **Xilinx**, *Vivado Design Suite User Guide High-Level Synthesis (UG902)*. Xilinx (v2016.1) User Guide, 2016*a*.

10. **Xilinx**, *Vivado Design Suite User Guide,Designing with IP (UG896)*. Xilinx (v2016.1) User Guide, 2016*b*.

11. **Xilinx**, *UltraFast High-Level Productivity Design Methodology Guide (UG1197)*. Xilinx (v2017.1) User Guide, 2017.

12. **Zepernick, H.-J.** and **A. Finger**, *Pseudo Random Signal Processing: Theory and Application*. Wiley, 2013.